Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2017

Verification of Well-formedness in Message-Passing Asynchronous Systems modeled as Communicating Finite-State Machines

Shiva Shankar Nalla

Iowa State University

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

Part of the Computer Sciences Commons

Verification of well-formedness in message-passing asynchronous systems modeled
as communicating finite-state machines

by

Shiva Shankar Nalla

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
Wensheng Zhang

Iowa State University

Ames, Iowa

2017

DEDICATION

I would like to dedicate this thesis to my major professor Dr.Samik Basu and my parents Mr. Ramprasad Nalla, Mrs. Sujatha Nalla for their constant moral support through out my Master's education at Iowa State University. I would also like to thank my friends in the VeriLast group for their guidance and support during conducting my research and writing of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

ABSTRACT

Asynchronous systems with message-passing communication paradigm have made major inroads in many application domains in service-oriented computing, secure and safe operating systems and in general, distributed systems. Asynchrony and concurrency in these systems bring in new challenges in verification of correctness properties. In particular, the high-level behavior of message-passing asynchronous systems is modeled as communicating finite state machines (CFSMs) with unbounded communication buffers/channels. It has been proven that, in general, state-space exploration based automatic verification of CFSMs is undecidable–specifically, reachability and boundedness problems for CFSMs are undecidable. In this context, we focus on an important path-based property for CFSMs, namely well-formedness–every message sent can be eventually consumed. We show that well-formedness is undecidable as well, and present decidable sub-classes for which verification of well-formedness can be automated. We implemented the algorithm for verifying the well-formedness for the decidable subclass, and present our results using several case studies such as service choreographies and Singularity OS contracts.

CHAPTER 1.   INTRODUCTION

Distributed Systems are becoming increasingly popular in most of the computing platforms and business areas, owing to its advantages with high processing power, increased reliability, huge storage and increased scalability. It has indeed become a basis for most of the modern applications. Some of its application areas include Telecommunications, Peer to Peer Networks, Parallel computation and Real-time processing. According to Coulouris et al. (2011), a distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages. Given its significant usage and applications in vast areas of business and technology, it becomes imperative to study and analyze the properties of such systems to avoid system failures and any unwanted behavior.

Message based communication is one of the widely used ways in which the hardware or software components in a distributed system communicate with each other. However, the peers or components in a system can sometimes become busy serving other requests and sometimes, there can be delay in the message transmission. Hence, the peers in a system should be tolerant enough in processing and communicating messages. This is usually achieved via asynchronous communication. Asynchronous systems with message-passing communication paradigm have made major inroads in many application domains–in service-oriented computing WS-CDL (2006), in secure and safe operating systems Singularity (2004); Fähndrich et al. (2006) and in general distributed systems Armstrong (2002). The Web Services Choreography Description Language (WS-CDL) is an XML based language that defines and describes peer-to-peer collaborations of participating peers and their observable behavior in which communication is achieved via message exchanges. The specification is targeted at providing collaborations between peers irrespective of the hosted platforms and programming models of the individual peers. Singularity is aimed at building architecture, services and applications that are highly dependable. One of

the important features of Singularity OS is process isolation to guarantee that a process cannot access or corrupt code or data of another process. The inter-process communication is achieved via message passing. Armstrong (2002) proposes a new scheme of Universal Binary Format (UBF) for exchange of messages in distributed systems instead of the traditional method of XML based communication scheme.

At a high-level, the dynamics of such systems are captured using communicating finite-state machines (CFSM) Brand and Zafiropulo (1983), where the behavior of each component referred to as machine, peer or agent is described using a finite state machine. The exchange of messages between each of these machines is performed through channels. One of the ways in which a channel can be seen is as a FIFO queue where the size of the queue indicates the maximum number of messages that the channel can hold before the receiver is ready to consume. In this interpretation, it can also indicate the maximum delay that the channel can induce on a message. A channel can also be represented as a queue with no ordering of messages involved and a message can be consumed randomly if it is available in the queue. In this work, we consider FIFO ordering of messages in the queue. As per the CFSM description, the peer evolves from one state to another via a directed transition resulting from production (sending) or consumption (receiving) of message(s). A sent-message is appended to the channel which is a buffer or queue of the receiver, and the receiver consumes a message if it is available in its channel. Hence, a system can be seen as a set of communicating finite state machines, in which each machine communicates with the other by sending a message to the channel of the receiving machine.

1.1   Verification of Well-formedness in an asynchronous communication

Given a model of the asynchronous system and a set of properties, model checking determines whether the given model satisfies the specified properties. However, if the state space of the system is not finite, model checking is undecidable in general owing to the infinite state space. One of the primary factors influencing the size of the state space in asynchronous communication is the size of the channels used by the peers; which is not known a priori. In Brand and Zafiropulo (1983), the authors established that two important decision problems related to

CFSMs (with channels modeled as queues) is undecidable—boundedness: does the size of queues in the CFSMs always remain within some finite bound, and reachability: can one compute the reachable state-space of any CFSMs. This makes automatic verification of many important properties including deadlock-freedom undecidable. In Yu and Gouda (1982); Gouda et al. (1987), the authors further discuss the boundaries of decidability of deadlock-freedom based on the type and number of buffers in communication paradigm. On the other hand, the authors, in Gouda and Chang (1986); Gouda et al. (1984), considered liveness properties described in terms of infinitely often occurrences of states in the machines. These properties are undecidable, in general, as they depend on the state-space computation as well. As a result, research has focused on identifying either a sub-class of CFSMs (i.e., sub-class of message-passing asynchronous systems) for which verification becomes decidable, or the sufficient conditions for deciding satisfiability of properties, or a class of property, whose conformance is still automatically verifiable in general asynchronous systems.

In our thesis, we define the property "Well-formedness", which states that every message that is produced should be eventually consumed; i.e as a system evolves by means of sending a message M from a peer P_1 to another peer P_2; the system should also be able to eventually reach a future configuration where the message M is consumed by P_2. As the peers in an asynchronous system communicate over unbounded receive-queues, it is not guaranteed that every message sent by a peer will be eventually consumed by the receiving peer. This may be because the receiving peer is never ready to consume the message, which may result in deadlock and/or lead to an unwanted behavior of the asynchronous system. Hence, well-formedness becomes one of the important property for asynchronous communication to ensure a safe behavior in its evolution.

However, verification of well-formedness can be challenging. Well-formedness check requires that at every configuration in the state space, every peer in the current configuration will be able to consume the messages in its queue in the near future. However, the size of the state space or computation of the reachable state space was previously proved to be undecidable by Brand and Zafiropulo (1983). So, that leaves us with the question on whether verifying well-formedness for asynchronpus systems is decidable or undecidable. And how does the results

vary for various types of asynchronous communication; for example, the messages in the queue can be ignored or deferred by the receiving peer Desai et al. (2013) or the messages can be consumed by the receiving peer in a random order.

We prove that verifying well-formedness property in asynchronous systems is undecidable in general. The proof relies on simulation of Turing machine by asynchronous CFSMs and reduction of the famous Halting problem to the problem of verifying well-formedness in asynchronous systems.

We also identify two important sub-classes of asynchronous systems for which well-formedness can be automatically tested. These sub-classes have been extensively studied in the existing literature in the context of message-passing systems in different domains, and are determined in terms of the observable interactions or sequence of send actions. The consumption of messages (receives) are typically viewed as local to the receiving peers and are not considered observable. Hence, in Web service choreography language, the behavior of the services are often described in terms of the messages being sent by the participating peers/services WS-CDL (2006). Similarly, the specifications of desired interactions in Singularity OS communication contracts Singularity (2004) and UBF(B) communication contracts in distributed Erlang programs Armstrong (2002) are described using interactions between (messages being sent by) participating entities.

One of the sub-classes is referred to as synchronizable Fu et al. (2005). Systems are synchronizable if and only if the interactions resulting from asynchronous communication over unbounded queues can be captured by the interactions resulting from synchronous communication (where every sent-message is immediately consumed). We prove that synchronizable systems are well-formed. Another sub-class (a superset of synchronizable sub-class) is referred to as k-send-bounded. Systems are k-send-bounded if and only if the interactions resulting from asynchronous communication over unbounded queues can be captured by the interactions resulting from asynchronous communication over k-bounded queues. We prove that for systems in k-send-bounded sub-class, well-formed property can be automatically verified. Membership (of asynchronous systems two peers) in synchronizable and k-send-bounded sub-classes has been shown to be decidable in Basu and Bultan (2011, 2014).

1.2 Contributions

1. **Decidability of well-formedness property** We will prove that verification of well-formedness property in asynchronously communicating CFSM is undecidable. The proof involves the construction of asynchronous system that simulates a Turing machine and on the reduction of halting problem to the testing of the system's well-formedness.

2. **Identifying sub-classes of systems** We present two interesting sub-classes of asynchronous systems for which well-formedness can be automatically tested: Synchronizable sub-class and k-send-bounded sub-class. Checking whether an asynchronous system over two peers belongs to one of these sub-classes is already proven to be decidable Basu and Bultan (2011, 2014)

3. **Case Studies** We present the experimental results of running our implementation technique on various service contracts like Reservation Session, Metaconversation, etc. and Singularity OS contracts like TpmContract, TcpContract and KeyboardContract.

1.3 Outline

The rest of the thesis is organized as follows. Chapter 2 presents various formalisms for asynchronous systems as communicating finite state machines. Chapter 3 discusses the contributions of this work in light of the existing work in verification of properties in asynchronous message-passing systems. Chapter 4 discusses the Turing machine simulation and presents the proof of undecidability for the well-formedness property. Chapter 5 discusses and presents the well-formedness characteristics for the identified sub-classes. Chapter 6 discusses the result of applying our technique on several case studies. Finally, Chapter 7 summarizes the future avenues of research.

CHAPTER 2.   BACKGROUND

This chapter discusses various formalisms and definitions for asynchronous systems modeled as communicating state machines and their behavior in the presence of bounded buffers and unbounded buffers. We illustrate the behavioral semantics using examples of asynchronous systems. These definitions follow the ones presented in Basu and Bultan (2014) in the context of developing sub-classes of asynchronous systems for which automatic verification is feasible.

2.1   Peers & Systems

A peer or machine in an asynchronous system is represented as a communicating Finite State Machine (CFSM). The communication between the peers is modeled as sending or receiving of messages between the CFSMs. Each state in a peer is capable of doing a send (produce) action or receive (consume) action or both. A send action would result in addition of a message to the receiving peer's buffer and a receive action would result in consumption of a message from its buffer. We formally define the Peer behavior in 1.

Definition 1 (Peer Behavior) *A peer behavior (or simply a peer), denoted by $\mathcal{P}$, is a finite state machine (M, S, s_0, δ) where M is the union of input (M^{in}) and output (M^{out}) message sets, S is the finite set of states, $s_0 \in S$ is the initial state, and $\delta \subseteq S \times (M \cup \{\epsilon\}) \times S$ is the transition relation.*

A transition $\tau \in \delta$ can be one of the following three types:

1. *a send-transition of the form $(s_1, !m_1, s_2)$ which sends out a message $m_1 \in M^{out}$,*

2. *a receive-transition of the form $(s_1, ?m_2, s_2)$ which consumes a message $m_2 \in M^{in}$ from its input queue, and*
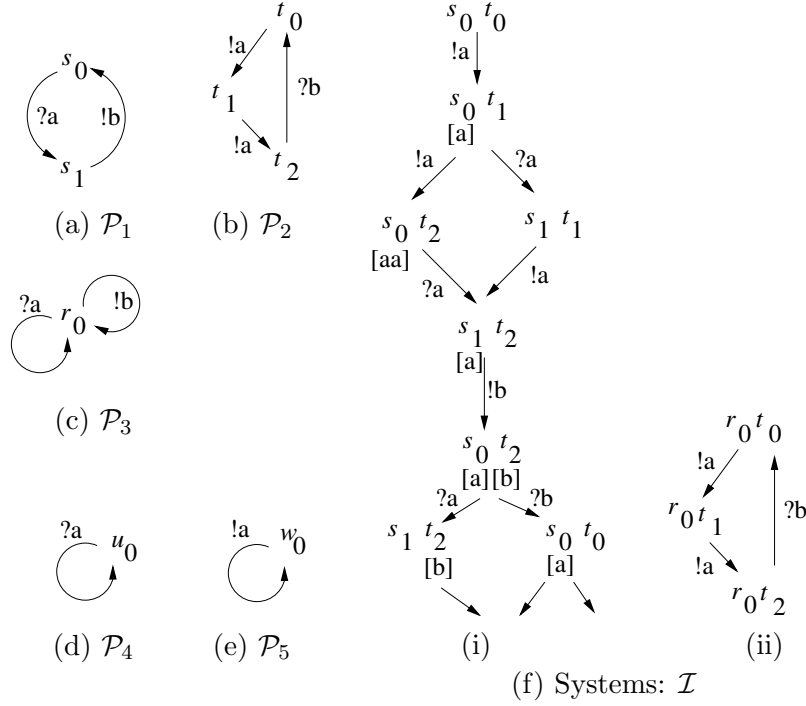
Figure 2.1 (a–e) Peers; (f-i) Partial View of $\mathcal{I}$ composed of $\mathcal{P}_1, \mathcal{P}_2$; (f-ii) $\mathcal{I}_0$ composed of $\mathcal{P}_3$, $\mathcal{P}_2$

3. *an ϵ-transition of the form (s_1, ϵ, s_2). We write $s \xrightarrow{a} s'$ to denote that $(s, a, s') \in \delta$.*

We will focus on deterministic peer behaviors, where $\forall s_1, s_2 : s \xrightarrow{a} s_1 \ \wedge \ s \xrightarrow{a} s_2 \Rightarrow (s_1 = s_2)$. Peer behaviors can be made deterministic by following standard methods for translation of non-deterministic state machines to deterministic ones. Figure 2.1 presents CFSM behavioral representation of peers. The initial states are sub-scripted with 0.

Figures 2.1(a, b) presents CFSM behavioral representation of two peers $\mathcal{P}_1$ and $\mathcal{P}_2$. The initial states are sub scripted with 0.

An asynchronous system usually consists of multiple peers. The system behavior is composed of message exchanges between these peers in the system. We now formally define System Behavior in 2

Definition 2 (System Behavior) *A system behavior (or simply a system) over a set of peers $\langle \mathcal{P}_1, \ldots, \mathcal{P}_n \rangle$, where $\mathcal{P}_i = (M_i, S_i, s_{0i}, \delta_i)$ and $M_i = M_i^{in} \cup M_i^{out}$, is denoted by a state machine*

(possibly infinite state) $\mathcal{I} = (M, C, c_0, \Delta)$, *where M is the set of messages, C is the set of configurations, c_0 is the initial configuration, and Δ is the transition relation defined as:*

1. $M = \cup_i M_i$

2. $C \subseteq S_1 \times \mathcal{Q}_1 \times S_2 \times \mathcal{Q}_2 \times \ldots \times S_n \ldots \mathcal{Q}_n$ *such that* $\forall i \in [1..n] : \mathcal{Q}_i \subseteq (M_i^{in})^*$

3. $c_0 \in C$ *such that* $c_0 = (s_{01}, \epsilon, s_{02}, \epsilon, \ldots, s_{0n}, \epsilon)$*; and*

4. $\Delta \subseteq C \times M \times C$,

   for $c = (s_1, q_1, s_2, q_2, \ldots s_n, q_n)$ *and* $c' = (s'_1, q'_1, s'_2, q'_2, \ldots s'_n, q'_n)$

   (a) $c \xrightarrow{!m} c' \in \Delta$ *if* $\exists i, j \in [1..n] : m \in M_i^{out} \cap M_j^{in}$,

      (i) $s_i \xrightarrow{!m} s'_i \in \delta_i$, (ii) $q'_j = q_j m$,

      (iii) $\forall k \in [1..n] : k \neq j \Rightarrow q_k = q'_k$ *and* (iv) $\forall k \in [1..n] : k \neq i \Rightarrow s'_k = s_k$

   (b) $c \xrightarrow{?m} c' \in \Delta$ *if* $\exists i \in [1..n] : m \in M_i^{in}$

      (i) $s_i \xrightarrow{?m} s'_i \in \delta_i$, (ii) $q_i = m q'_i$,

      (iii) $\forall k \in [1..n] : k \neq i \Rightarrow q_k = q'_k$ *and* (iv) $\forall k \in [1..n] : k \neq i \Rightarrow s'_k = s_k$

   (c) $c \xrightarrow{\epsilon} c' \in \Delta$ *if* $\exists i \in [1..n]$

      (i) $s_i \xrightarrow{\epsilon} t'_i \in \delta_i$, (ii) $\forall k \in [1..n] q_k = q'_k$ *and* (iii) $\forall k \in [1..n] : k \neq i \Rightarrow s'_k = s_k$

In summary, a state in the system is described by the local states of the participating peers and their respective receive queues. At each transition, one of the peers can update its local state by sending or receiving messages. The messages sent are added to the tail of the receive queue of the receiver, while messages consumed are removed from the head of the receive queue. The sending of the messages are never blocked. The receiving of messages can be blocked if the message to be consumed is not present at the head of the queue.

Figure 2.1(f-i) presents a partial view of a system $\mathcal{I}$ resulting from the asynchronous communication between peers $\mathcal{P}_1$ and $\mathcal{P}_2$. The messages in the respective queues of the participating peers is enclosed by [.], where the left-most entity is at the head of the queue. Each action of sending or receiving a message results in a new transition. Each state in such

a system is referred to as a *configuration*. The initial configuration of the system has the local start states from each of the participating peers, in this example, s_0t_0. s_0 cannot make a transition as it is waiting for message a to be sent by t_0 in $\mathcal{P}_2$. $\mathcal{P}_2$ makes a transition from t_0 to t_1 by sending message a to $\mathcal{P}_1$. The new resulting configuration is $s_0t_1[a][]$. Note that in its current configuration, either $\mathcal{P}_1$ can consume the message resulting in a new configuration $s_1t_1[][]$ or $\mathcal{P}_2$ can make a send transition resulting in the configuration $s_0t_2[aa][]$. As discussed earlier, the send transitions are never blocked and the system will be able to evolve by sending messages as long as there are send transitions available with the local states in the individual peers. Note that, the system in this example has unbounded number of configurations as every time $\mathcal{P}_2$ completes a cycle in its local transitions, it adds the message a to $\mathcal{P}_1$'s buffer twice; however, $\mathcal{P}_1$ can consume only one a every time in the cycle of its local transitions. Thus, the number of messages pending to be consumed by $\mathcal{P}_1$ can increase in an unbounded fashion (production of messages outruns the consumption of messages) and eventually leads to an infinite state space.

In the above example, we have seen an asynchronous system with unbounded buffer space that has the capability of growing its space space by sending messages in an unbounded fashion. The buffer space available for the peers in an asynchronous system can also be bounded in order to limit the behavior of the system. Here, we provide a formal definition for a $k - bounded$ system in 3

Definition 3 (k-bounded System Behavior) *A k-bounded system (denoted by $\mathcal{I}_k$) is a system where the receive queue capacity for any peer is at most k. The k-bounded system behavior is, therefore, defined by augmenting condition 4(a) in Definition 2 to include the condition $|q_j| < k$, where $|q_j|$ denotes the length of the queue for peer j.*

In k-bounded system $\mathcal{I}_k$, the send actions are blocked when the corresponding receive queue, where the sent message is supposed to be buffered, is full (i.e., it already contains k messages pending to be consumed by the receiver). Therefore, $\mathcal{I}_k$ has a finite state-space. Hence, in the above example Figure 2.1(f-i), $\mathcal{P}_2$ can no longer send messages to $\mathcal{P}_1$ once $\mathcal{P}_1$'s receive buffer is filled with k a's. $\mathcal{P}_2$'s send actions will be blocked until $\mathcal{P}_1$ consumes a message from its buffer which provides buffer space for $\mathcal{P}_2$ to send messages to $\mathcal{P}_1$.

The above definition 3 also gives us an interesting observation. The number of configurations in a system $\mathcal{I}_{k+1}$ (bound $k+1$) can be more than the number of configurations in $\mathcal{I}_k$; i.e. $\mathcal{I}_{k+1}$ can have additional configurations along with the configurations that are present in $\mathcal{I}_k$. Let's consider the systems $\mathcal{P}_1$ and $\mathcal{P}_2$ from the example Figure 2.1(f-a,f-b,f-i). Consider systems $\mathcal{I}_2$ and $\mathcal{I}_1$. $\mathcal{I}_2$ having a buffer bound of two can make a send transition from the configuration $s_0 t_1[a][]$ to a new configuration $s_0 t_2[aa][]$. $\mathcal{I}_1$ having a buffer bound of 1 cannot reach this configuration as the send transition by $\mathcal{P}_2$ will be blocked as $\mathcal{P}_1$'s buffer is already full in $s_0 t_1[a][]$. However, it should be noted that $\mathcal{I}_{k+1}$ contains all the configurations that are available in $\mathcal{I}_k$ as in the above example.

Definition 4 (Synchronous System Behavior) *A synchronous system over a set of peers* $\langle \mathcal{P}_1, \ldots, \mathcal{P}_n \rangle$, *where* $\mathcal{P}_i = (M_i, S_i, s_{0i}, \delta_i)$ *with* $M_i = M_i^{in} \cup M_i^{out}$ *and* $\forall i, j : i \neq j, M_i^{in} \cap M_j^{in} = M_i^{out} \cap M_j^{out} = \emptyset$, *is denoted by a finite state machine* $\mathcal{I}_0 = (M, C, c_0, \Delta)$, *where* M *is the set of messages,* C *is the set of states,* c_0 *is the initial state, and* Δ *is the transition relation defined as*

1. $M = \cup_i M_i$

2. $C \subseteq S_1 \times S_2 \times \ldots \times S_n$

3. $c_0 \in C$ *such that* $c_0 = (s_{01}, s_{02}, \ldots, s_{0n})$

4. $\Delta \subseteq C \times M \times C$ *for* $c = (s_1, s_2, \ldots s_n)$ *and* $c' = (s_1', s_2', \ldots s_n')$ $c \xrightarrow{!m} c' \in \Delta$ *if*

   $\exists i, j \in [1..n] : m \in M_i^{out} \cap M_j^{in}$, *such that* $s_i \xrightarrow{!m} s_i' \in \delta_i$ *and* $s_j \xrightarrow{?m} s_j' \in \delta_j$; *and*
   $\forall k \in [1..n] : k \notin i, j \Rightarrow s_k' = s_k$

In a synchronous system, a sender can only send a message when the receiver is ready to consume the message, and the act of sending and receiving messages is synchronized. Figure 2.1(f-ii) presents a synchronous system. Each transition is labeled with the send action; however, note that for synchronous systems, the corresponding receive action also happens with the same transition. In Figure 2.1(f-ii), every send transition implies a send transition followed by an immediate receive (consumption) transition.

2.2 System Behavior modeled as a language

The interaction between peers in an asynchronous system can be modeled as a sequence of message exchanges. The sequence of message exchanges lets the system evolve into different configurations. Below, we provide a formal notation of modeling the interactions in a system as language and also define Language Equivalence.

Definition 5 (Language Model and Language Equivalence) *Given a system $\mathcal{I} = (M, C, c_0, F, \Delta)$ over a set of peers $\langle \mathcal{P}_1, \ldots, \mathcal{P}_n \rangle$, a path π is a finite or infinite sequence of configuration of the form*

$$c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} c_2 \xrightarrow{a_3} \ldots$$

where $a_i \in M$. We say that m is a send if $\exists k \colon !m = a_k$. Furthermore, m_j is the j-th send in the path π if $!m_j = a_i$ and there are $j - 1$ sends in π's prefix: $c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} c_2 \ldots \xrightarrow{a_{i-1}} c_{i-1}$.

Given a path π in a system $\mathcal{I} = (M, C, c_0, F, \Delta)$, a send sequence in π is $m_1 m_2 m_3 \ldots$ such that m_j is the j-th send in the π.

The language of $\mathcal{I} = (M, C, c_0, F, \Delta)$, denoted by $\mathcal{L}(\mathcal{I})$, is the set of sequences of send actions on any (finite or infinite) path in $\mathcal{I}$. Systems $\mathcal{I}$ and $\mathcal{I}'$ are languageequivalent if and only if $\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}')$. Some examples of send sequences for the system in Figure 2.1(f-i) are aabaabaab.., ababaa.., ababab...

When we consider bounded buffer systems, the send actions are usually restricted by the capacity of the receiver buffer and as a result, the behavior of a system with smaller capacity buffers is smaller (in terms of sequences of send actions) than the ones with larger capacity buffers. Formally, we say that

Proposition 1 $\forall k \geq 0 : \mathcal{L}(\mathcal{I}_k) \subseteq \mathcal{L}(\mathcal{I}_{k+1}) \subseteq \mathcal{L}(\mathcal{I})$.

Based on the notion of language equivalence and synchronous behavior discussed above 4 and 5, synchronizability is defined as follows.

Definition 6 (Synchronizability) *A system $\mathcal{I}$ over a set of peers is said to be synchronizable when its language resulting from the asynchronous composition of peers is identical to the language resulting from the synchronous composition of the same set of peers. That is, $\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}_0)$.*

When a system is synchronizable, its properties (with respect to send actions) can be automatically verified by verifying the same properties for the corresponding synchronous system, whose behavior can be expressed in finite state-space. In Basu and Bultan (2011), the authors proved that synchronizability is a decidable property and presented an efficient method for the decision procedure.

Proposition 2 (Deciding Synchronizability (Basu and Bultan (2011))) *A system $\mathcal{I}$ is synchronizable if and only if $\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1)$.*

The above results hold for two-peer asynchronous systems. Note that, $\mathcal{I}_0$ and $\mathcal{I}_1$ are finite state machines and, therefore, their language equivalence can be automatically determined. While synchronizability is an important sub-class of asynchronous systems (it was reported that over 90% of Singularity OS channel contracts are synchronizable), there are still asynchronous systems, which are not synchronizable, but whose behavior may be represented using bounded capacity queues. In Basu and Bultan (2014), the authors presented the condition under which asynchronous system's behavior (with respect to send-sequences) can be captured using finite capacity queues. Note that, this does not imply boundedness of asynchronous system (which is undecidable); instead, our results simply state that send-sequences of an asynchronous system with unbounded queues can be captured using finite capacity queues. Once the condition is satisfied, one can compute the necessary queue capacity using the following proposition. The following is generalized version of the above proposition in the context of two-peer asynchronous system.

Proposition 3 (Finding k-send-boundedness (Basu and Bultan (2014))) $\forall k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}) \Leftrightarrow \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$.

When a system $\mathcal{I}$ is k-send-bounded, then the properties in terms of send-sequences can be verified by verifying the same properties using $\mathcal{I}_k$.

CHAPTER 3.   LITERATURE REVIEW

In this chapter, we discuss some of the important works that have been done previously in modeling and auto verification of properties in asynchronous systems. Different approaches were taken to address the verification problem in asynchronous systems. As the verification problem for these systems is undecidable in general, researchers have focused on identifying sub-classes of the systems and in some cases, sufficient conditions which when satisfied, makes the verification of these properties decidable.

3.1   Asynchronous Systems as Communicating finite state machines

Brand and Zafiropulo (1983) models processes in an asynchronous communication as explicit communicating finite state machines and uses implicit queues to represent channels; the queues are modeled to have unbounded capacity in order to support channels with arbitrary varying capacities. The processes communicate by sending messages to one another over the channels. The protocol model used by the work defines the properties of the system in terms of *execution* of the model which is a sequence of global states, where each global state represents the current state of all individual processes and their queue content. The execution starts from the initial global state which contains all the initial states of each process. The execution proceeds further in terms of steps where each step involves either sending a message or consumption of a message. In our work we refer to these each of these global states as a *configuration*. A global state is reachable if there exists a sequence of steps starting with the initial global state that eventually leads to the target global state. Also, a protocol can be deadlocked if it eventually reaches a global state where the channels of all processes are empty and there are no transmissions from

any individual peers in that global state. All the states that are reachable were referred to as N-tuple stable states, where N represents the number of processes.

The work focuses on a problem statement which is to find all the executable receptions and all the stable states for a given system. An executable reception is a pair (s, x), s is a state in a peer and x is a message to be consumed, leads to a reachable global state. It proves that finding the reachable state space and identifying all the stable states is undecidable in general. Further, it also discusses that boundedness, whether or not the queue-sizes in all the reachable configurations in a CFSM remains within some finite bound is undecidable; however, when the channels are bounded, finding the global reachable state space and stable states is decidable. It also shows that the problem is decidable for systems having two processes when any one channel is unbounded. But it unsolvable if any system with more than 2 processes has any channel unbounded. The proposed solution consists of considering N executions of N process separately. It builds N trees, one for each process that represents all possible executions for each process. Each state that is reached multiple times in a tree is given a different name; similarly, each message that is transmitted or received more than once uses a different name to distinguish different possible states and possible transitions in this tree protocol. The conditions for communicating CFSMs to be identified as a tree protocol can be referred from Brand and Zafiropulo (1983).

The solution approach reduces the original problem statement into 2 subproblems: first, to solve the same problem for tree protocols and second, to decide when to terminate the growing of trees, without missing any stable states or possible receptions. The work Brand and Zafiropulo (1983) solves the first subproblem; however, the second subproblem is unsolvable in general and a partial solution is proposed. The partial solution works by checking if further expansion of tree uncovers new receptions or stable states in the general graph. The solution marks the states dead that satisfies the type 0 or type 1 dead criteria. The type 0 dead criteria is similar to the methods used in perturbation method used in West (1978); it ignores any newly generated global state that is already previously generated. The type 1 dead criteria states that the sub-tree of a state t_i can be ignored if any message sent from t_i can only be received at dead states. The expansion of the tree terminates when new transmissions can

only be appended below the dead states. The limitation of the approach is that termination is not guaranteed for all protocols with unbounded channels; they guarantee termination only in bounded channels and also in systems with 2 processes with only one unbounded channel. In other cases, the solution recommends the use of a natural parameter by the user based on the protocol that limits the search for a solution and also mentions it as one of the future work areas.

3.2 Verification via Synchronizability

We now discuss some of the works that have tried to address the verification problem using Synchronizability property. Simply stated, if the interaction behavior of the peers in a system remains the same when asynchronous communication is replaced with synchronous communication, it is said to satisfy the Synchronizability property. A formal definition for Synchronizability property is provided at 6. Below we discuss the works and results of Basu and Bultan (2011), Fu et al. (2004), Fu et al. (2005).

Fu et al. (2004) discusses the techniques for analyzing interactions of composite web services specified in BPEL (Business Process Execution Language) that interact through XML messages. The work presents the tool set framework that translates BPEL specifications to intermediate representation consisting of guarded automata consisting of unbounded queues made of XPath expressions which is finally translated to a verification language. For verification, they use Promela (SPIN model checker) as the language. As SPIN is a finite state verification tool, it can allow for a partial verification by bounding the queue size. The work addresses the problem of partial verification by introducing synchronization which checks that if the conversation set remains same when asynchronous communication is replaced with synchronous communication, then the composite web service is synchronizable. The work proposes three necessary and sufficient conditions for a composition to be synchronization which are Synchronous compatibility, Autonomy and Loss-less composition. Synchronous compatibility states that any synchronous composition constructed does not contain a state where a peer p_i is ready to send message to peer p_j but p_j is not ready to consume the message. Autonomy states that in a composite system, every state in a peer can have only one type of transitions : either send or receive

including states reachable via ϵ transitions; else the state is a final state or reaches a final state through ϵ transitions. Loss-less composition states that if a Cartesian product of the peers is constructed by making the initial states in each peer as final states; then the projection of the Cartesian product to each peer should be equivalent to the original skeleton where the initial state is marked as final state. The authors prove in this paper that if the above three conditions are satisfied, then the composite system is synchronizable.

Now we will discuss work done in Fu et al. (2005). This work focuses on conversations which are the global sequence of messages exchanged among web services. The paper attempts to check whether a LTL property will be satisfied by the conversations of a composite web-service. It shows that this problem is undecidable when dealing with systems with unbounded queues and proposes synchronizability technique in order to solve it. If a composite web-service falls into synchronizable subclass, then the conversation set remains the same when the asynchronous communication is replaced with synchronous communication. The authors present sufficient conditions that guarantee synchronizability and also show that synchronizability analysis can be used to check the realizability of top-down conversation specifications. They initially prove that synchronous communication is always a subset of asynchronous communication by checking the synchronization send sequences; the converse does not hold true. The conditions proposed are the synchronous compatibility and Autonomous conditions that are discussed earlier in Fu et al. (2004). The complexity of checking whether both the conditions are satisfied is exponential in the size of the peers. Then, they prove that if a composite web-service satisfies both these conditions, then for every conversation generated by the composite system, there exists a run having immediate receive actions followed by send actions that generate the same conversation and that the composite web-service is synchronization.

The conditions proposed in this paper only ensure that the languages resulting from asynchronous interactions and synchronous counterpart are similar. Below we further discuss Basu and Bultan (2011) that proposes conditions and criteria which ensures that any property, either linear or branching time satisfied by the asynchronous composite system is also satisfied by its synchronous counterpart. Also, the autonomy condition can be easily violated by systems when a peer contains a state with both send and receive actions out of it.

Basu and Bultan (2011) tries to address the problem of Choreography conformance, which is to identify if a set of given services adhere to a given choreography specification. The problem falls in the area of service oriented computing and is undecidable in general; and the work tries to identify a class of asynchronous systems for which this problem can be efficiently checked; this class of systems must follow the synchronization property. The results of this work could be applied to various types of asynchronous systems that interact via message based communication such as Singularity Channel contracts and BF contracts where message interactions need to adhere to given contract specifications. The Synchronizability problem has been an open problem for several years and the authors in this paper provide an algorithm to determine the synchronization of a set of asynchronously communicating peers with unbounded queues and hence, solves dependability of synchronization problem. The algorithm works by comparing the behavior of peers with synchronous communication and peers with asynchronous communication bounded by a queue size of one. Since both of them result in finite state spaces, the comparison can be easily done using finite state verification tools. The authors prove that if the interaction behavior of peers are same for synchronous and one-bounded asynchronous communication, then it is also the same for unbounded asynchronous communication.

The paper further discusses different variations of conditions for synchronizability based on different types of expressivity of the choreography specification. Some examples are if the choreography specification is expressed using some desired sequencing of send actions as linear time temporal logic or regular expressions, then synchronizability is based on language equivalence; however, if it expresses using desired branching of send actions as in CTL or ACTL or ACTL*, then it is based on bisimulation equivalence or simulation equivalence. It provides definitions and notions of language equivalence, Simulation Pre-order, Bisimulation equivalence and relates them with synchronizability. Refer to Basu and Bultan (2011) for definitions. Bisimulation equivalence ensures that the branching behavior of interactions between two systems is same and is also known that it preserves all temporal logic properties including branching time temporal logic properties Clarke et al. (1999). It follows that if a system is bi-simulation synchronizable, then it is also language synchronizable; hence, providing a more strict notion to verify synchronizability. In summary, the verification of conformance

of a system to a choreography specification is addressed by - bi simulation synchronizability works for specifications expressed in any temporal logic; simulation synchronizability works for specifications expressed in universal fragment of temporal logic and language synchronizability works for specifications expressed as FSA or LTL temporal logic properties.

3.3   General boundedness in Asynchronous Systems

We now discuss another interesting property in asynchronously communicating systems: Unboundedness. When peers in an asynchronous communication interact, they store their messages pending to be consumed in their queues. Verification of properties in such systems is challenging as the message queues can grow arbitrarily large and often lead to infinite state space behavior making auto verification of properties undecidable. Many works handle this problem by bounding the queues in some cases and identifying sub-classes of systems for which verification become decidable or propose verification techniques that are sound but incomplete.

We will initially discuss the work of Cécé and Finkel (2005) that deals with verification of communicating finite state machines with unbounded channels. Specifically, it considers the analysis of infinite half-duplex systems made of finite systems communicating over unbounded channels. The half-duplex property for two machines and two channels says that each reachable configuration should have at most one channel non-empty. The authors prove that half-duplex systems have a recognizable reachability set. It then shows that the symbolic representation of this reachability set can be computed in polynomial time and uses the results to solve several other verification problems. It also proves that membership in half-duplex class is decidable. Further, it shows that it becomes Turing powerful for a more generalized system with more than 2 machines and proves that the verification of Propositional Linear Time Logic (PLTL) or Computation Tree Logic (CTL) properties for these systems is undecidable.

The paper initially discusses concepts around finite state machines, then define seven problems that are of specific interest: the reachability problem, the deadlock problem, the unspecified reception problem, the executable transition problem, the weak boundedness and the strong boundedness problem and the recognizable computation problem. It then discusses half-duplex systems consisting of two machines and two channels; it shows that each reachable

configuration of the half-duplex systems of two machines is reached by a specific execution that is used in computing the channel-recognizable reachability set. The idea is to split the sequence of actions leading to a configuration into 2 sets: a). $1 - bounded$ execution which ensures that that the number of messages available in both the channels is atmost one and b).send actions that use only one machine and one channel. This type of transmission is always available for half-duplex systems with two machines and two channels and finally, they obtain a channel- recognizable set. The authors then prove that the reachability set can be computed in polynomial time and deduce that the seven verification problems mentioned above are also decidable for half-duplex systems of two machines. It presents the conditions to be checked for systems to be half-duplex and then proves that the half-duplex property is decidable in polynomial time. Though a recognizable representation of reachability set could be constructed in polynomial time, it shows that verifying properties expressed in CTL and PLTL for these half-duplex systems is undecidable. Finally, it shows that for half-duplex systems having more than two machines, the systems become Turing powerful and computing the reachability set is also undecidable for such machines.

Basu and Bultan (2014) proposes necessary and sufficient conditions under which asynchronously communicating systems with unbounded queues (let's call it as A) exhibit behavior that is equivalent to the those systems with finitely bounded queues (let's call it as B). The behavior is modeled as a global sequence of send actions over the communicating peers. When the condition holds, A becomes automatically verifiable as B will have a finite state space because of bounded queues leading to it becoming automatically verifiable and all the verification results produced for B still remain valid for A. Also, once the condition is satisfied, the work proposes to find the bound by iteratively checking the interaction behavior equivalence between systems with queue sizes k and $k + 1$ starting with $k=1$. The paper starts with defining formalisms of asynchronous systems and their behavior, then defines $k - boundedsystem$ and language equivalence which we have used in our current work.

In this paper, the properties are modeled as temporal ordering of send sequences expressed in LTL. The authors provides the first condition and prove that if a composite asynchronous system with buffer bound of k and another with same peers and unbounded queues are language

equivalent, then the systems with buffer bound of k and $k + 1$ are also language equivalent and it holds both ways. The second condition is that for two asynchronous systems I and $I^{'}$, if $\mathcal{L}(I)$ = $\mathcal{L}(I^{'})$; then for any LTL property Ψ over the send actions, I satisfies $\Psi \Leftrightarrow I^{'}$ satisfies Ψ. The third condition is the major one which when actually satisfied guarantees the existence of bound k such that $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})$. The authors define an unbounded send sequence which if satisfied in any configuration of system $\mathcal{I}$ (system with unbounded queues), then the interaction behavior cannot be represented by the peers in any $\mathcal{I}_k$ (k-bounded system); as it leads to the receiving peers to have an infinite receive queue. Similarly, if a peer can consume all the messages from its buffer to reach state $s_{i'}$ from state s_i, and s_i is not send-simulated by $s_{i'}$, then also it requires the peer to have infinite receive queues. Hence, if a composite system can be verified that they do not satisfy these above two qualities, then there exists a bound. The paper presents an algorithm that checks for above qualities and also provide proof of correctness for the algorithm. The paper claims the results of this technique can be applied on wide variety of composite systems and presents the results of the experimental runs. The major contribution of this paper is that it present the sufficient conditions which when satisfied by a composite system with unbounded queues has the same interaction behavior as of that of a bounded system that allows for the auto verification of the properties in the unbounded system.

3.4   Restricted Communication Models

In addition to the above properties and verification techniques discussed so far, several techniques have been previously used to handle the state space explosion problem in verification of concurrent systems. Some approaches include partial order reduction methods (POR), program slicing, data abstraction and state compression technique. However, to reduce the generality of the systems, certain works have also focused on restricted communication models. One of the interesting area is the domain specific approach that leverages the restrictions imposed by the programming domain in order to achieve greater reductions. We now discuss one such class of works in the context of verification of MPI (Message Passing Interface) programs. Below we discuss the works and contributions of Siegel (2005), Manohar and Martin (1998) and Vakkalanka et al. (2010) in this area.

Siegel (2005) focuses on the domain of parallel programs that employ MPI programs. The paper tries to verify halting properties, properties specific to states in a program where execution halts. Some halting properties are freedom from deadlock and assertion of values in variables after the termination of a program. It discusses certain basic semantics of the MPI programs. The function MPI_SEND is used to send a message to another process, one must specify the destination process and a message tag to use it. The function MPI_RECV is used to receive messages. This function can also specify its source process, or may use the wild card value MPI_ANY_SOURCE which indicates that the message can be accepted from any source process. It may also specify the tag of the message it wishes to receive, or may use the wild card value MPI_ANY_TAG. A receive operation that uses either or both of these wildcards is called a wildcard receive.The use of wildcards and tags allows for a great flexibility in how the messages are selected for reception into a process. Prior works have already proved that if a program does not contain wild card receives, then for a model M of the program, M is deadlock free $\Leftrightarrow$ no synchronous execution of M can deadlock and there was no state explosion problem. The state explosion problem occurs because of the need to represent all message channels in the presence of wild card receives.

An MPI program basically consists of a fixed number of concurrent processes each executing its own code, with no shared variables and communicate only through the MPI functions. Unlike several other programs that usually bound their channels to block send actions, the MPI standard does not bound its channels. The send action may block at any time unless the receiving process is at a state from which it can receive the sent message synchronously. A receive statement is blocked if there are no messages that match the statement parameters. A state in a model for a concurrent system can be potentially halted if either of the send, receive or synchronous actions are available. This work shows that the hypothesis on wild card receives may be relaxed to allow the use of MPI_ANY_TAG and still be able to verify different properties. It also expands the range of properties to include all halting properties. Further, it provides a model checking algorithm that handles MPI_ANY_SOURCE by moving back and forth between a synchronous and buffering mode as state space progresses. The authors claim that though the

approach is similar to the partial order reduction (POR) method, the number of states explored is much less and the method also works in certain cases where POR techniques do not work.

Manohar and Martin (1998), on the other hand presents conditions under which the slack of a channel in a distributed system can be changed without affecting its behavior, and such a distributed system is said to be slack elastic. The paper also discusses some program transformations that can be used in the design in the concurrent systems the correctness of which depends on the conditions presented. Vakkalanka et al. (2010) extends the work of Manohar and Martin (1998) and Siegel (2005) further by focusing on adding buffering into the MPI sends with out introducing deadlocks or other safety assertion violations. While adding additional buffer usually improves the performance of the programs; if the programs contain non-deterministic behavior such as a wild card receive as discussed earlier in Siegel (2005), then it can lead to issues as deadlocks or assertion violations. This paper introduces Lamport's happens-before relation which is generally used in studying concurrency and partial order semantics for MPI programs and provides a precise characterization of slack elasticity based on the formulation of the happens-before relation. It discusses on how to identify the code patterns of the culprit sends in slack inelastic programs that has the potential to increase non-determinism in the program and introduce new bugs. The authors present this algorithm to identify culprit sends incorporated into their dynamic verifier that can verify larger MPI programs.

Now we discuss another interesting area that has been significantly focusing on interactions between asynchronously communicating systems: session types. Session types are used in modeling communication centered applications, where the interacting set of peers follow behavior specified in the protocols. One of the basic observations in such communication centered applications is that they exhibit a highly structured sequence of interactions involving branching and recursion, which as a whole form a session. This structure of a conversation is abstracted as a type using operational syntax and then used as a basis for validating and verifying programs that fall in those disciplines. Some of the properties of the session types being are: a).Interactions within a session never induce any communication error referred to as communication safety, b). Channels are used linearly and are deadlock free in a single session referred to as linearity and

progress, and c). the communication sequences in a session follow the specifications declared in the session type referred to as session predictability. Below we discuss some of the works that have been focusing on using session types for verification.

We will initially discuss the work Bocchi et al. (2013) that focuses on assuring safe interactions in large-scale distributed systems. Previous methods were based on centralized verification or restricted specification methods and have limited applicability. This work proposes monitored $\pi-$calculus with dynamic usage of multiparty session types that enables safety assurance of asynchronously communicating distributed components. The framework allows for both static and dynamic verification of components. Asynchrony in the system was added through the means of explicit routers and global queues. The basic idea was to capture the decentralized nature of distributed application development and provide better support for heterogeneous distributed systems by allowing components to be independently implemented, using different languages, libraries and programming techniques; allow for being independently verified, either statically or dynamically and also retain the strong global safety properties of statically verified homogeneous systems. The authors initially discuss various formalisms related to multiparty session types, global types and local types with assertions to build the protocols and then provide the interaction semantics for processes over a network in the presence of session environment. The work also introduces a behavioral theory over monitored networks that allows com positional reasoning over trusted and untrusted components.

On the other hand, Honda et al. (2008) focuses on extending the work on binary session types involving two processes to multiparty asynchronous sessions. Multiparty session types (MPST) is a typing discipline for communication programming that was originally developed in the πcalculus for verification of distributed software. When communication between multiple peers (more than two) is abstracted as binary sessions between every pair of communicating peers, the abstraction tends to lose essential sequencing information for the interactions and hence, the whole conversation needs to be represented as a single session. The work presents two major challenges involved in extending it to multiparty systems: duality and linearity analysis. Unlike in binary sessions, where a peer behavior is a dual type to the other peer's behavior, when there are multiple peers involved, the whole conversation cannot be constructed from

behavior of one single peer. Similarly, linearity analysis of channels that ensure safety and progress is another challenge because with the combination of multiparty and asynchrony, a conflict of actions can arise more easily and correct sequencing of interactions among these multi-peers need to be ensured. The authors introduce global types that can abstract intended conversation structure among multiple parties and is also used a basis of efficient type checking through its projection onto individual peers. The work discusses the fundamental properties of session types such as communication safety, progress, session fidelity and establishes them for the multiparty asynchronous interactions.

3.5   Progress and Liveness in communicating Finite State Machines

In this section, we will discuss some of the previous works that have focused on liveness and progress properties in CFSMs. On these lines, we discuss the works and results of Gouda et al. (1984), Gouda and Chang (1986) and Deniélou and Yoshida (2012).

Gouda et al. (1984) focuses on the general problem of communication progress between two finite state machines and discusses its relationship to properties such as boundedness, freedom from deadlocks and unspecified receptions. It tries to find if for any two finite state machines that exchange messages through two uni-directional channels, whether there exists a positive integer K such that the communication between these peers over K-capacity channels is guaranteed to progress indefinitely. In such a communication, all the states with buffer size less than or equal to K for both peers, referred to as K-reachable states are expected to be progress states. A progress state is one that is not deadlocked, free from unspecified reception and not an overflow state. The authors prove that the problem of finding whether such bound K exists is undecidable. They also present a practical class of systems called as alternating communicating machines for which the problem is shown to be decidable. In an alternating communicating machine, each sending edge is followed only by a receiving edge and as a result, its buffer size is always bounded by two. It can also be seen that it is a subclass of half-duplex machines for 2 peers discussed in the previous section. The authors also provide two sufficient conditions that can ensure indefinite progress - a) Compatible communication with no mixed nodes (every node in a peer can either send or receive but not both) and b) Deadlock freedom

for an abstracted system with reduced number of message types ensures the original system is also deadlock-free.

Gouda and Chang (1986) on the other hand focuses on defining and proving liveness properties for networks of communicating finite state machines. Liveness of a node in a machine is usually measured by the occurrence of the respective node infinitely often during the course of communication, which necessitates a fairness strategy in how the peers communicate. The authors define three degrees of fairness: Node, Edge and Network fairness. A communication sequence is said to be node-fair if and only if every node that is enabled infinitely often is executed infinitely often; edge-fair if and only if every edge that is enabled infinitely often is executed infinitely often; network-fair if and only if every node or edge that is encountered infinitely often is executed infinitely often. Every edge-fair sequence is node-fair and every network-fair sequence is edge-fair. The authors further define three degrees of node liveness based on the respective fairness assumption. They provide sufficient conditions to verify liveness under each of the above fairness assumptions that are also shown to be effective when the system is composed of infinite reachable states. Unlike some of the earlier works that construct a temporal or classic system for proving liveness for general systems, this work proves the liveness for a network of communicating finite state machines by constructing a closed cover graph based on the directed graph representations of the individual peers in the system. On similar lines, Deniélou and Yoshida (2012) focuses on multiparty systems relating them to the communicating finite state machines and defines a subclass of CFSMs called as Multiparty Session Automata (MSA) that automatically satisfies distributed safety and progress properties. It also presents a new type system for multiparty session mobile processes and proves that typed processes conform to the safety and liveness properties defined in CFSMs.

Thus, as we have seen from the existing work of different contributions, they have not directly considered whether or not messages are always consumed. In some of the existing work Gouda and Chang (1986) and Gouda et al. (1984), progress and/or liveness property have been considered, which includes consumption of messages as one of the desired behavior. However, the liveness properties described for the systems are in terms of state-space of the system.

In contrast, our work solely focuses on sequences of message exchanges to define well-formed property.

CHAPTER 4. DECIDABILITY OF WELL-FORMED PROPERTY IN ASYNCHRONOUS SYSTEMS

In this chapter, we discuss the characterizations of well-formed behavior, fairness requirement and then prove our result that the well-formed property for asynchronous systems with unbounded queues is undecidable. In 4.1, we define well-formed behavior and discuss its characterizations along with the fairness requirement. In 4.2, we construct an asynchronous system that can simulate a Turing machine; we then show that it is undecidable by reducing the Halting problem to the testing of system's well-formedness.

4.1   Well-formed Behavior

As noted in the Definition 2, the peers in an asynchronous system communicate by sending and receiving messages over unbounded receive-queues. As a result, in general, it is not guaranteed that every message sent by one peer will be eventually consumed by the receiver. This may be because the receiver is never ready to consume the message, which may result in deadlock and/or lead to unwanted behavior of the asynchronous system. We define the well-formed behavior as follows.

Definition 7 (Well-formed Behavior) *An asynchronous system is said to be well-formed if and only if along all evolutions of the system, every message sent is eventually consumed.*

We will denote a well-formed system $\mathcal{I}$ ($\mathcal{I}_k$) as $\text{WF}(\mathcal{I})$ ($\text{WF}(\mathcal{I}_k)$, resp.). An important aspect come into consideration in the context of well-formed behavior. The well-formed behavior is a liveness behavior and liveness conformance in distributed/concurrent systems rely on the concept of *fairness*.

In a typical distributed system, scheduling of peers depends on the computing environment where the system is deployed. Fairness, in this context, refers to the fairness of the scheduler and its associated scheduling algorithm. Fairness amounts to ensuring or imposing the condition that each peer is scheduled infinitely often, if the peer has some enabled moves. We will refer to this as *global* fairness or simply fairness. For instance, under fairness, the system resulting from $\mathcal{P}_4$ and $\mathcal{P}_5$ (Figures 2.1(d, e)) is well-formed. On the other hand, consider the system resulting from $\mathcal{P}_2$ and $\mathcal{P}_3$ (Figures 2.1(b,c)). Even under global fairness, this system is not well-formed. This is because the branch in $\mathcal{P}_3$ that corresponds to consuming a may not be scheduled. One can further refine the notion of fairness to include the constraint that every enabled branch in every peer must not be ignored by the scheduler forever. We refer to this as *global-local* fairness. We will consider well-formedness in the context of global-local fairness.

Verifying well-formedness. In order to precisely describe the condition for well-formedness, we will use the following notations. For a system $\mathcal{I} = (M, C, c_0, \Delta)$ with n peers $\langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n \rangle$ and a configuration $c = (s_1, q_1, s_2, q_2, \ldots, s_n, q_n)$ of the system, we use

$$c{\downarrow}^{st} = (s_1, s_2, \ldots, s_n) \quad \text{projection of configuration to local states}$$

$$c{\downarrow}^{st}_{\mathcal{P}_i} = s_i \quad \text{projection of configuration to } \mathcal{P}_i\text{'s local state}$$

$$c{\downarrow}^{qu}_{\mathcal{P}_i} = q_i \quad \text{projection of configuration to receive queue of } \mathcal{P}_i$$

For any peer $\mathcal{P}$, $\texttt{Path}(s)$ denotes the set of paths starting from the state s present in the behavior of $\mathcal{P}$. Each path is described using a sequence of send or receive actions in the path. For instance, in Figure 2.1, for s_0 in $\mathcal{P}_1$, $\texttt{Path}(s_0) = \{\langle ?a \ !b \ ?a \ !b \ldots\rangle\}$; for r_0 in $\mathcal{P}_3$, $\texttt{Path}(r_0) = \{\langle ?a \ ?a \ ?a \ldots\rangle, \ \langle ?a \ !b?a \ !b \ldots\rangle, \ \ldots\}$. For any path π in a peer, we denote the sequences of receives in that path as $\texttt{recvseq}(\pi)$. For instance, for any $\pi \in \texttt{Path}(s_0)$, the $\texttt{recvseq}(\pi) = \langle a \ a \ a \ \ldots\rangle$.

For any queue, the messages in the queue can be consumed by the sequence of receive actions by the appropriate peer, where the ordering of the messages in the queue matches with the ordering of the corresponding receives in the sequence of receive actions. The sequence of messages in a queue is denoted by $\texttt{seq}(q)$ (for empty queue, the sequence is ϵ).

Therefore, determining well-formedness of a system will require verifying that for every configuration c of the system reachable from its start configuration, the following condition holds for all i: if $c\downarrow^{qu}_{\mathcal{P}_i} = q_i \neq \emptyset$, then $\texttt{seq}(q_i)$ is a prefix of $\texttt{recvseq}(\pi)$ for some $\pi \in \texttt{Path}(c\downarrow^{st}_{\mathcal{P}_i})$. In other words, for every non-empty queue, the peer responsible for consuming messages from that queue must have a path to consume the messages. For instance, in the system (Figure 2.1(f-i)), the queue corresponding to the peer $\mathcal{P}_1$ contains $[aa]$ at the configuration $c := s_0[aa]t_2[\ ]$. The above condition is satisfied because: $\texttt{seq}(c\downarrow^{qu}_{\mathcal{P}_1}) = \langle a\,a \rangle$ and it is prefix of the receive sequence $\texttt{recvseq}(\pi)$, where $\pi = \texttt{Path}(s_0)$.

In the following, we will prove that well-formed behavior is undecidable, in general. We will reduce the halting problem to verifying well-formed behavior. The reduction is based on constructing an asynchronous system that simulates a Turing machine. The construction is identical (modulo extensions for well-formed behavior) to the one presented in Akroun et al. (2016) and follows from the one discussed in Finkel and McKenzie (1997).

4.2 Asynchronous Systems simulating TM

We use the standard definition of Turing machine–a finite state machine that moves from one state to another as it reads and updates the symbols residing on a tape using a head (pointer to a tape location). It is a tuple $TM = (Q, q_0, \Delta, \Sigma, \{L, R, \_\})$, where Q is a finite set of states, $q_0 \in Q$ is the start state and $\Delta \subseteq (Q \times \Sigma) \to (Q \times \Sigma \times \{L, R, \_\})$ is a deterministic transition relation. We will write $(q, a) \to (q', a', L)$ to denote a transition, where the machine is at state q, it reads the symbol a from the tape and evolves to state q' but replacing a with a' at the tape location and moving the head to the left.

We consider a Turing machine TM which on accepting a string w halts; otherwise, it rejects the string and initiates a loop. The loop proceeds by moving the head to the right. We will assume that the alphabet Σ contains special symbols b (blank symbol), $\triangleright$ (start-marker) and $\#$ (end-marker). We will also assume that the string w contains symbols in $\Sigma \backslash \{b, \triangleright, \#\}$ and is delimited with the start-marker $\triangleright$ and the end-marker $\#$.

We will use two communicating peers $\mathcal{P}_1$ and $\mathcal{P}_2$ to capture the TM behavior. The message exchanged by the peers correspond to the symbol set Σ. Following Akroun et al. (2016), we will

use the superscript 1 to denote messages sent by $\mathcal{P}_2$ to be consumed by $\mathcal{P}_1$ and the superscript 2 to denote the messages sent by $\mathcal{P}_1$ to be consumed by $\mathcal{P}_2$. We also add a message h^1 (to be consumed by $\mathcal{P}_1$, sent by $\mathcal{P}_2$) and h^2 (to be consumed by $\mathcal{P}_2$ and sent by $\mathcal{P}_1$). Furthermore, we will add message *accept* that can be sent by $\mathcal{P}_1$ and consumed by $\mathcal{P}_2$; and a message *halt* that can be sent by $\mathcal{P}_1$ and not consumed by $\mathcal{P}_2$. The reason for the inclusion of h^is, *accept* and *halt* will be explained below.

Our construction of an asynchronous system relies on capturing the state-space of the Turing machine using the peer $\mathcal{P}_1$ and replicating the corresponding tape contents using $\mathcal{P}_1'$s queue. Therefore, for every state $q \in Q$ of TM, there is a corresponding state s_q in $\mathcal{P}_1$. The tape is modeled by the queue contents as follows. If the tape contains the strings u and v, where u is to the left of the head and the head is pointing to the first symbol of v, then the queue contents of $\mathcal{P}_1$ are arranged as uhv, i.e., the head is represented by a message h in the queue.

Construction of $\mathcal{P}_2$.   The peer $\mathcal{P}_2$ has the following transition behavior:

$$s_0 \xrightarrow{!\triangleright} s_1 \xrightarrow{!h^1} s_2 \xrightarrow{!m_1^1} \ldots \xrightarrow{!m_n^1} s_{n+1} \xrightarrow{!\#^1} s_{relay} \qquad \text{(Trans-}\mathcal{P}_2\text{-init)}$$

where s_0 is the start state of $\mathcal{P}_2$, the input string to the machine M is $w = m_1 m_2 \ldots m_n$ and $\triangleright, \#$ are the start- and end-markers for the string. It is immediate that the $\mathcal{P}_2$ sets up the queue for $\mathcal{P}_1$ in such a way that the queue content replicates the tape of TM with the head (represented by the message h^1) ready to "read" the first symbol of the input string w.

From the state s_{relay}, the $\mathcal{P}_2$ consumes any message other than *accept* sent to it and relays it back to $\mathcal{P}_1$.

$$s_{relay} \xrightarrow{?m^2} s_{relay'} \xrightarrow{!m^1} s_{relay} \quad \forall m_i \in \Sigma \cup \{h^2\} \qquad \text{(Trans-}\mathcal{P}_2\text{-relay)}$$

On the other hand, on consuming *accept*, $\mathcal{P}_2$, moves from s_{relay} to s_{end_2}, from where it consumes all pending messages (from Σ) in the queue.

$$s_{relay} \xrightarrow{?accept} s_{end_2} \xrightarrow{?m} s_{end_2} \quad \forall m \in \Sigma \cup \{h^2\} \qquad \text{(Trans-}\mathcal{P}_2\text{-end)}$$

$$s_{q0} \xrightarrow{?h^1} . \xrightarrow{?\triangleright^1} . \xrightarrow{!\triangleright^2} . \xrightarrow{!h^2} s_{q1}$$

$$s_{q1} \xrightarrow{?h^1} s'_{q1}$$
$$s'_{q1} \xrightarrow{?\#^1} . \xrightarrow{!\#^2} . \xrightarrow{!h^2} s_{accept}$$
$$s'_{q1} \xrightarrow{?x^1} . \xrightarrow{!\triangleright^2} . \xrightarrow{!h^2} s_{q2}$$

$$s_{q2} \xrightarrow{?h^1} s'_{q2} \xrightarrow{?z^1} . \xrightarrow{!z^2} . \xrightarrow{!h^2} s_{q2}$$
where z is x or y

$$s_{q2} \xrightarrow{?z^1} . \xrightarrow{?h^1} . \xrightarrow{?b^1} . \xrightarrow{!h^2} . \xrightarrow{!z^2} . \xrightarrow{!b^2} s_{q3}$$
where z is x or y

$$s_{q3} \xrightarrow{?z^1} . \xrightarrow{?h^1} . \xrightarrow{?y^1} . \xrightarrow{!h^2} . \xrightarrow{!z^2} . \xrightarrow{!\#^2} s_{q4}$$
where z above is either x or y or b

$$s_{q4} \xrightarrow{?z1^1} . \xrightarrow{?h^1} . \xrightarrow{?z2^1} . \xrightarrow{!h^2} . \xrightarrow{!z1^2} . \xrightarrow{!z2^2} s_{q4}$$
where $z1$ is either x or y or b
and $z2$ is either x or y

$$s_{q4} \xrightarrow{?h^1} . \xrightarrow{?\triangleright^1} . \xrightarrow{!\triangleright^2} . \xrightarrow{!h^2} s_{q1}$$

$$s_{rej} \xrightarrow{?h^1} . \xrightarrow{?z^1} . \xrightarrow{!z^2} . \xrightarrow{!h^2} s_{rej}$$
where z is either x, y or b (blank)

$$s_{rej} \xrightarrow{?h^1} . \xrightarrow{?\#^1} . \xrightarrow{!h^2} . \xrightarrow{!b^2} . \xrightarrow{!\#^2} s_{rej}$$
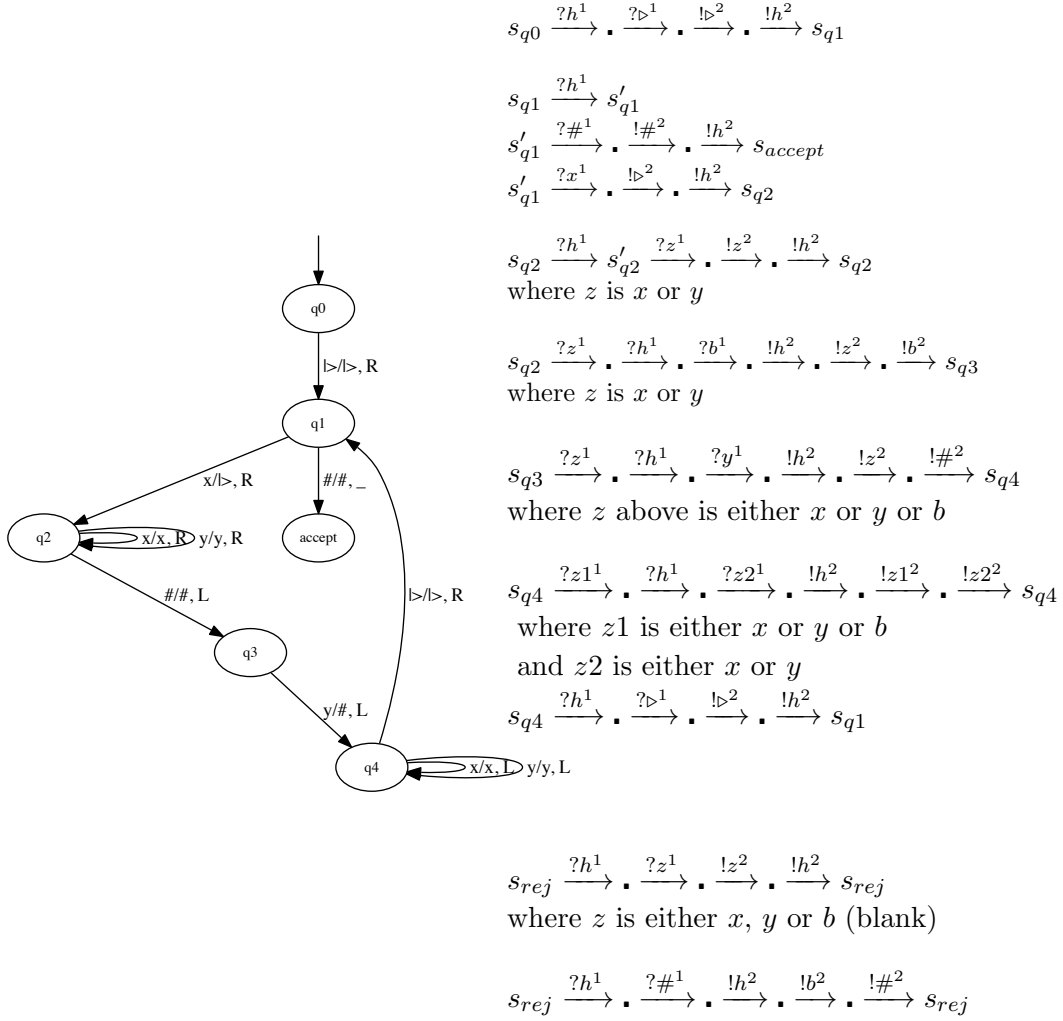
Figure 4.1   Turing Machine and the corresponding Peer $\mathcal{P}_1$

Construction of $\mathcal{P}_1$.   The peer $\mathcal{P}_1$ has five types of transition sequences. The first two types of transition sequences correspond to the transition relation of the Turing Machine TM. For the transition $(q, m) \to (q', m', R)$ in TM, $\mathcal{P}_1$ has

$$s_q \xrightarrow{?h^1} s_{q_1} \xrightarrow{?m^1} s_{q_2} \xrightarrow{!m'^2} s_{q_3} \xrightarrow{!h^2} s'_q \qquad \text{(Trans-}\mathcal{P}_1\text{-right)}$$

Similarly, for the transition $(q, m) \to (q', m', L)$ in TM, $\mathcal{P}_1$ has

$$s_q \xrightarrow{?n^1} s_{q_1} \xrightarrow{?h^1} s_{q_2} \xrightarrow{?m^1} s_{q_3} \xrightarrow{!h^2} s_{q_4} \xrightarrow{!n^2} s_{q_5} \xrightarrow{!m'^2} s_{q_3} \qquad \text{(Trans-}\mathcal{P}_2\text{-left)}$$

where $m, m', n \in \Sigma$. Furthermore, for every state q in TM, the corresponding state s_q in $\mathcal{P}_1$ has the following transition sequences (loop), which allows $\mathcal{P}_1$ to send the queue contents

(configuration of the tape) to $\mathcal{P}_2$.

$$s_q \xrightarrow{?m_i^1} s_{q_1} \xrightarrow{!m_i^2} s_q \quad \forall m_i \in \Sigma \qquad \text{(Trans-}\mathcal{P}_1\text{-relay)}$$

The above three transition sequences of $\mathcal{P}_1$ along with the transition sequences in $\mathcal{P}_2$ collectively captures the evolution of TM, where $\mathcal{P}_1$ starts with the current configuration (state of TM and next tape-configuration to $\mathcal{P}_2$, and $\mathcal{P}_2$ relays back the next tape-configuration to $\mathcal{P}_1$, which appears in the queue corresponding to $\mathcal{P}_1$.

When the TM initiates a loop (input string w is not accepted by TM), it moves the head to the right. The peer-moves simulate the movement of the head in the TM to the right till it reaches the end-marker $\#$. At this point, on reading the $\#$, a new blank symbol is added before $\#$. This captures the "extension" of the tape as the TM moves (in a loop) the head to the right of the tape. The transition sequence representing this starts from s_q, where q in a state in TM such that TM moves to state q when it does not accept the input string.

$$s_q \xrightarrow{?h^1} s_{q_1} \xrightarrow{?\#^1} s_{q_2} \xrightarrow{!h^2} s_{q_3} \xrightarrow{!b^2} s_{q_4} \xrightarrow{!\#^2} s_q \qquad \text{(Trans-}\mathcal{P}_1\text{-blank)}$$

When $\mathcal{P}_1$ moves to a state s_{accept} corresponding to the accepting state in TM, it sends $!accept$ to $\mathcal{P}_2$, moves to s_{end_1} from where it consumes all pending messages and sends a $halt$ message to $\mathcal{P}_2$. That is,

$$s_{accept} \xrightarrow{!accept} s_{end_1} \xrightarrow{?m} s_{end_1} \xrightarrow{!halt} s_{halt} \quad \forall m \in \Sigma \cup \{h^1\} \qquad \text{(Trans-}\mathcal{P}_1\text{-end)}$$

Figure 4.1 presents a TM that halts on the input string of the form $\triangleright x^n y^n \#$. From any state of the form q_i, the TM moves to a reject state (rej) if the head reads an unexpected symbol. From the reject state, the TM moves the head to the right forever. The corresponding transition system for $\mathcal{P}_1$ is presented on the right-hand side. From the s_{rej} state (corresponding to the state of the TM from where it moves the head to the right forever), the peer $\mathcal{P}_1$ simulates right move of the head till it reads $\#$. At this point, a new blank symbol is added to the left of $\#$. ($\mathcal{P}_1$ also includes the moves from the s_{accept}, as presented in Trans-$\mathcal{P}_1$-end.)

When $\mathcal{P}_1$ and $\mathcal{P}_2$ interacts, $\mathcal{P}_2$ never consumes the message $halt$. Therefore, whenever TM moves to accepting state and halts, the peers simulating its behavior leads to a system

configuration which is not well-formed (existence of message *halt* that is never consumed by $\mathcal{P}_2$). On the other hand, if TM loops (does not accept the input string w), the peer-interaction is well-formed. This is because from all states except s_{accept}, $\mathcal{P}_1$ can move only by consuming messages sent to it by $\mathcal{P}_2$, which, in turn, relays all messages sent to it. In other words, we have reduced the halting problem to well-formed verification problem. If the system is well-formed, then the TM must not halt (does not accept the input string); otherwise, TM halts (by accepting the input string).

Theorem 4.2.1 *It is undecidable to verify whether an asynchronous message-passing system is well-formed.*

It follows from the above simulation and reduction of the halting problem in Turing machines that verification of well-formedness property in message-passing asynchronous systems is undecidable.

CHAPTER 5.   VERIFYING WELL-FORMEDNESS FOR SUBCLASS OF ASYNCHRONOUS SYSTEMS

In chapter 4, we have proved that verifying well-formedness property for asynchronous systems is undecidable. In this chapter, we present two subclasses of asynchronous systems for which well-formedness property can be automatically verified. One of the subclass is the k-bounded subclass discussed in Basu and Bultan (2014). Another subclass is the Synchronizability subclass discussed in Basu and Bultan (2011). In section 5.1, we will discuss the equivalence properties for a k-bounded subclass and provide the well-formedness properties that hold when an asynchronous system is k-send bounded; we also prove that well-formed behavior is auto verifiable for this subclass. In section 5.2, we prove that when an asynchronous system is synchronizable, then it is well-formed.

5.1   Deciding Well-formed behavior for Subclass of Asynchronous Systems

As noted in Proposition 3, the condition under which a two-peer system behavior can be captured using finite capacity queues is, $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1}) = \mathcal{L}(\mathcal{I})$. First, we prove that when $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$, the verification result of well-formed behavior of $\mathcal{I}_k$ holds for $\mathcal{I}_{k+1}$ as well. Then, we use the 3 result to prove that the well-formed behavior of $\mathcal{I}_k$ holds for $\mathcal{I}$

Theorem 5.1.1 $\forall k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1}) \implies \mathrm{WF}(\mathcal{I}_k) \iff \mathrm{WF}(\mathcal{I}_{k+1})$.

Proof 1 *First, assume that $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$ and $\mathrm{WF}(\mathcal{I}_k)$, but $\neg\mathrm{WF}(\mathcal{I}_{k+1})$. This implies that there exists a path π^{k+1} in $\mathcal{I}_{k+1}$*

$$c_0^{k+1} \xRightarrow{!m_0} c_1^{k+1} \xRightarrow{!m_1} \ldots c_n^{k+1} \xRightarrow{!m_n} c_{n+1}^{k+1} \ldots \qquad (\pi^{k+1}\text{-path})$$

such that m_n is not consumed along any evolution of the system starting from c_{n+1}^{k+1}. In the above $\overset{!m}{\Longrightarrow}$ denotes zero or more receive actions followed by !m action.

As $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$, the sequence of send actions $m_0 m_1 \ldots m_n \ldots$ is also present in the behavior of $\mathcal{I}_k$, which, in turn, implies the existence a path π^k in $\mathcal{I}_k$, where m_n is always eventually consumed along fair paths in the system. That is,

$$c_0^k \overset{!m_0}{\Longrightarrow} c_1^k \overset{!m_1}{\Longrightarrow} \ldots c_n^k \overset{!m_n}{\Longrightarrow} c_{n+1}^k \ldots \qquad (\pi^k\text{-path})$$

m_n *is consumed starting from c_{n+1}^k.*

WLOG, consider that $\mathcal{P}$ is responsible for consuming m_n and all other peers are denoted by $\mathcal{P}'$. Therefore, in π^{k+1}, $\mathcal{P}$ moves along a path where it reaches a state (in configuration c_{n+1}^{k+1}) from where it cannot consume m_n. On the other hand, in π_k, $\mathcal{P}$ never reaches such a state. As the peer behaviors are deterministic and the same sequence of send actions are present in both π^k and π^{k+1}, $\mathcal{P}$ must have a choice point of the form:

$$s \overset{!m}{\longrightarrow} s_1 \overset{?m'}{\longrightarrow} s_2 \text{ is branch 1 and } s \overset{?m'}{\longrightarrow} s_3 \overset{!m}{\longrightarrow} s_4 \text{ is branch 2.}$$

and it moves along branch 1 in one path and along branch 2 in the other; otherwise, both $\mathcal{I}_k$ and $\mathcal{I}_{k+1}$ would be not well-formed. Note that, as the same sequence of send actions are present in both π^{k+1} and π^k, the $\mathcal{P}'$ can move along the same path in π^{k+1} and π^k.

If the message sequence in paths π^{k+1} and π^k contain m followed by m', and $\mathcal{P}$ must move along branch 1 in both π^k and π^{k+1}. This is because, m cannot be sent before m' is sent (some other peer) along branch 2. Therefore, $\mathcal{I}_k$ (as $\mathcal{I}_{k+1}$ is not well-formed) is not well-formed violating our assumption.

If the message sequence in paths π^{k+1} and π^k contain m' followed by m, and $\mathcal{P}$ moves along branch 1 in π^k (and $\mathcal{P}$ moves along branch 2 in π^{k+1}), then the same sequence can be generated in another π'^k in $\mathcal{I}_k$, where $\mathcal{P}$ moves along branch 2 (as in π^{k+1}). This will violate well-formed behavior in $\mathcal{I}_k$, thus, contradicting our assumption as well.

Finally, if the message sequence in paths π^{k+1} and π^k contain m' followed by m, and $\mathcal{P}$ moves along branch 2 in π^k (and $\mathcal{P}$ moves along branch 1 in π^{k+1}), then there also exists different sequence generated by a path π'^k in $\mathcal{I}_k$, where m is followed by m', and $\mathcal{P}$ moves along

branch 1 (as in π^{k+1}). This will violate well-formed behavior in $\mathcal{I}_k$, thus, contradicting our assumption as well.

Therefore, $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1}) \Rightarrow (\text{WF}(\mathcal{I}_k) \Rightarrow \text{WF}(\mathcal{I}_{k+1}))$. Also, note that $\neg\text{WF}(\mathcal{I}_k) \Rightarrow \neg\text{WF}(\mathcal{I}_{k+1})$ for all k. This follows from the Proposition 1: any behavior in terms of send sequences in $\mathcal{I}_k$ is also present in $\mathcal{I}_{k+1}$.

This concludes the proof. □

Corollary 1 $\forall k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1}) \Rightarrow (\text{WF}(\mathcal{I}_k) \Leftrightarrow \text{WF}(\mathcal{I}))$.

Proof 2 *If $\neg\text{WF}(\mathcal{I})$, then there exists some n, such that $\neg\text{WF}(\mathcal{I}_n)$. If $n > k$ and $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$, then $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_n)$, which implies $\neg\text{WF}(\mathcal{I}_k)$ (from the above theorem). On the other hand, if $n < k$ and $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$, then $\neg\text{WF}(\mathcal{I}_n) \Rightarrow \neg\text{WF}(\mathcal{I}_k)$ because $\mathcal{L}(\mathcal{I}_n) \subseteq \mathcal{L}(\mathcal{I}_k)$ (from Proposition 1).* □

The above corollary, in essence, describes the subclass of two-peer message-passing asynchronous systems (k-send-bounded) for which determining well-formed behavior is decidable. In Basu and Bultan (2014), the authors have proved that the determining $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$ is decidable for two-peer asynchronous systems. This, in turn, allows us to decide well-formed behavior as well.

5.2   Well-formedness behavior for Synchronizable subclass

In this section, we will discuss on verification of well-formedness properties for Synchronizable systems. As discussed earlier, a system $\mathcal{I}$ over a set of peers is said to be synchronizable when its language resulting from the asynchronous composition of peers is identical to the language resulting from the synchronous composition of the same set of peers. We denote a synchronizable system as $\mathcal{I}_0$ as synchronous action do not need a buffer to store the sent messages. Systems belonging to synchronizable subclass have a definite behavior that every message that is sent is immediately consumed, i.e. a send action is always accompanied by a corresponding receive action unlike the general behavior where a message stays in the receiver's buffer waiting for the configuration to evolve into a state in which the message can be consumed. Figure 5.1 presents

an example of a Synchronizable system with two peers. In the diagram, only send actions are provided and it can be seen that every send action is always accompanied by a corresponding receive action which makes it a Synchronizable.
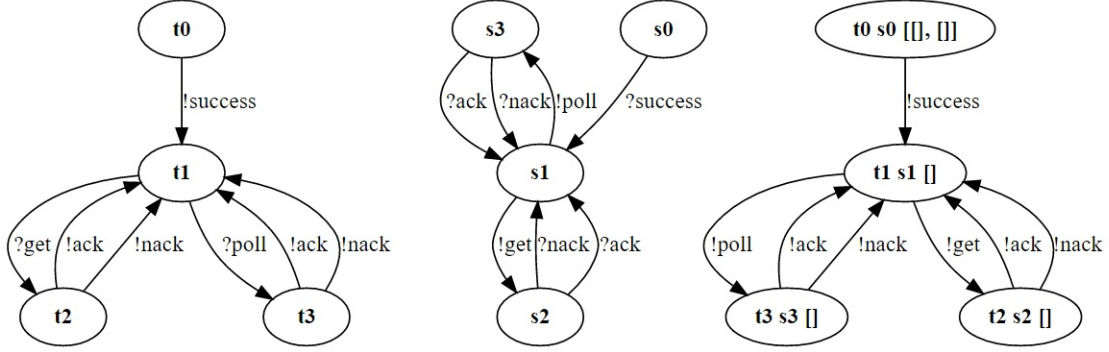

Figure 5.1   Example of a Synchronizable system

Below, we discuss the well-formedness properties for class of systems belonging to the Synchronizable subclass.

Corollary 2 (Guaranteed Well-formed Behavior) $\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1) \Rightarrow \text{WF}(\mathcal{I})$

Proof 3 *Using Corollary 1, we know that*

$$\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1) \Rightarrow \text{WF}(\mathcal{I}_0) \Leftrightarrow \text{WF}(\mathcal{I})$$

In $\mathcal{I}_0$, every message is consumed immediately. Therefore, it is immediate that $\mathcal{I}_0$ is well-formed, which, in turn, ensures that synchronizable system $\mathcal{I}$ is also well-formed. ☐

In short, synchronizable systems are well-formed and are a strict subclass of the k-send-bounded class.

CHAPTER 6.   IMPLEMENTATION

In this chapter, we discuss the implementation details including finding language equivalence, synchronizability and experiments to verify well-formedness behavior for various classes of asynchronous systems. We initially discuss about the tool and various components of the tool; then we present the experimental results on various case studies.

6.1   Tool Description

We have developed our tool using Java programming language with JDK 7. Figure 6.1 presents the architecture of the tool with its main components.
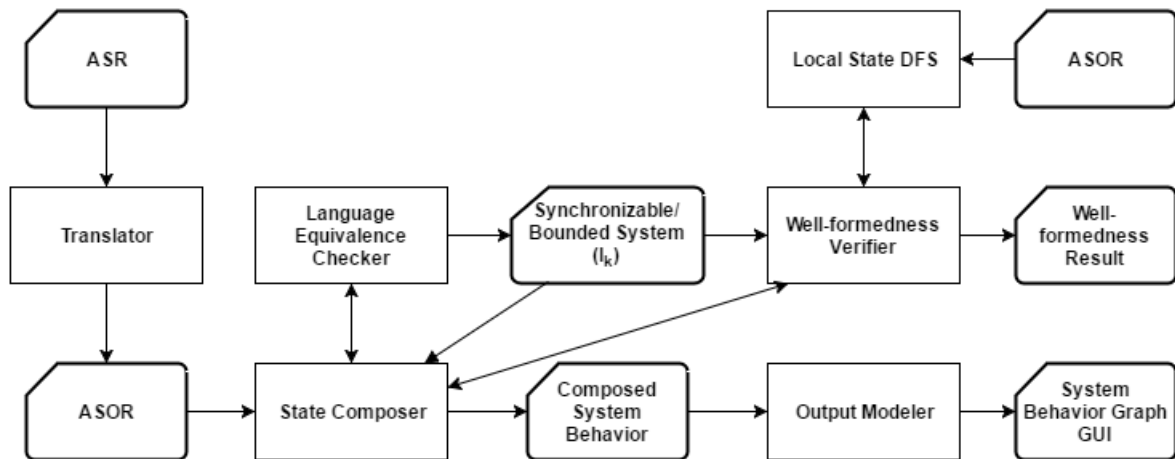

Figure 6.1   Tool Architecture

1. The tool initially takes as input, the asynchronous system representation that represents the properties and interactions in an asynchronous system. The input is parsed and

translated by the Translator module into an internal object representation that is used by other modules.

2. Language equivalence check is now performed on the ASOR using the State Composer to compute the next states. This step uses bi-simulation checking which has a stronger notion than Language equivalence as it conforms to a broader range of branching and temporal logic properties. The algorithm works in a depth-first fashion on the $\mathcal{I}_{k+1}$ by simultaneously checking that for every transition in $\mathcal{I}_{k+1}$, there are corresponding transitions in atleast one of its counter part $\mathcal{I}_k$ configurations. This module determines whether the system is synchronizable or k-send bounded or if the two systems are not bi-simulation equivalent, in which case we return the trace providing the necessary evidence.

3. Once we find that the system $\mathcal{I}_k$ is either synchronizable ($k=0$) or k-send bounded, we check if $\mathcal{I}_k$ is well-formed. The inputs for this module are the asynchronous system $\mathcal{I}$ and the value k that represents the bound found in the Language Equivalence module. For synchronizable systems, well-formedness is automatically implied. For systems that are not well-formed, a trace of the path that violates the well-formedness property is presented.

4. The tool also provides the composite system behavior of synchronizable or k-bounded systems in terms of a visualization graph.

Below is a description of each of the components in Figure 6.1.

1. **ASR (Asynchronous System Representation)** We use the language representation presented in the tool developed in Basu and Bultan (2014). The transitions are represented as "ptrans" relations and the start states are specified by "startPeer" relations.

```
%% reservation session

ptrans(p1, s0, in(req, p2), s1).

ptrans(p1, s1, in(cancel, p2), s2).
ptrans(p1, s1, out(succ, p2), s3).
ptrans(p1, s1, out(fail, p2), s4).
```

```
ptrans(p1, s2, out(cancelled, p2), s5).


ptrans(p1, s3, in(cancel, p2), s5).
ptrans(p1, s3, in(conf, p2), s5).


ptrans(p1, s4, in(cancel, p2), s5).


ptrans(p1, s5, in(reset, p2), s0).


startPeer(p1, s0).


%%% other peer


ptrans(p2, t0, out(req, p1), t1).


ptrans(p2, t1, out(cancel, p1), t2).
ptrans(p2, t1, in(succ, p1), t3).
ptrans(p2, t1, in(fail, p1), t4).


ptrans(p2, t2, in(cancelled, p1), t5).


ptrans(p2, t3, out(cancel, p1), t5).
ptrans(p2, t3, out(conf, p1), t5).


ptrans(p2, t4, out(cancel, p1), t5).


ptrans(p2, t5, out(reset, p1), t0).


startPeer(p2, t0).
```

Listing 6.1   Representation of Reservation session system in ptrans relations

Listing 6.1 gives the representation for reservation protocol. It consists of two peers communicating asynchronously. Each ptrans relation represents a transition which is either a send or receive. For example, $ptrans(p1, s0, in(req, p2), s1)$ indicates that there is a transition in peer $p1$ from state $s0$ to $s1$ by consuming message req sent by peer $p2$. The sending of message is indicated by $out(<m>, <p>)$ where the message m is sent

to peer p. Similarly, the consumption of message is indicated by $in(<m>, <p>)$ where the message m is received from peer p. The start state $s0$ in a peer p is indicated by $startPeer(p, s0)$. Figure 6.2 shows how the above representation is mapped to different states and transitions between the peers in the Reservation protocol.
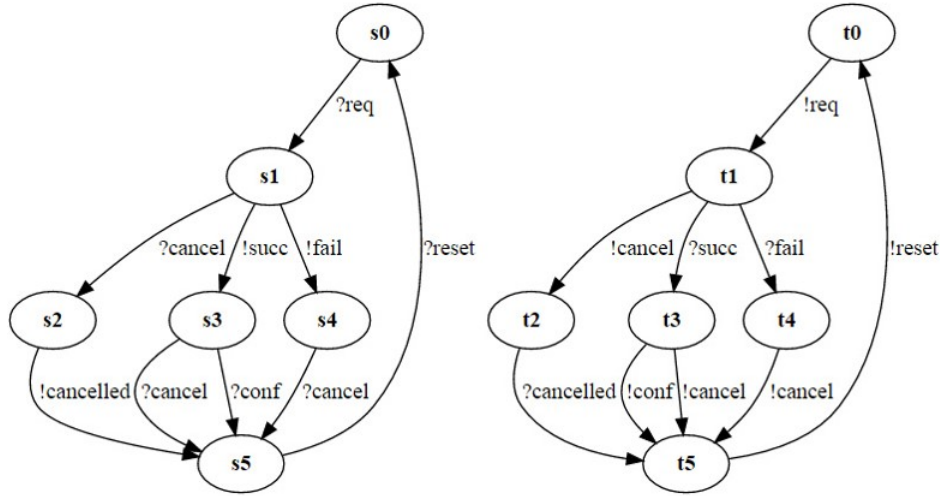

Figure 6.2   Peers and Transitions in Reservation Session

2. **ASOR (Asynchronous System Object Representation)** We parse the given ASR to get an internal representation of given asynchronous system in Java objects. Each state in a peer is represented as an object. A transition between two states in a peer is formulated as an edge object having references to source state, target state and associated message information. Then, each peer object consists of respective states and edges. A static list of peer objects is created and made accessible to the program globally.

3. **Translator** The main job of the Translator is to convert the given ASR that represents the asynchronous system into the ASOR format.

4. **State Composer** The state composer computes the next state for a given state configuration. It will use the information from the current state configuration including the local states of each peer, its corresponding message queues and the information from ASOR to

determine the next state. It is used by the modules Language Equivalence Checker and Well-formedness Verifier for generation of next states.

5. **Language Equivalence Checker** This module is used to determine whether the languages produced by two systems with bounds k and $k + 1$, i.e $\mathcal{I}_k$ and $\mathcal{I}_{k+1}$ are equivalent. As checking for language equivalence is computationally expensive, we use a polynomial $bi-simulation equivalence$ checking algorithm which is a stronger equivalence than language equivalence. The algorithm starts by checking whether the start configurations of the systems $\mathcal{I}_k$ and $\mathcal{I}_{k+1}$ are bi-simulation equivalent. It then recursively computes the bi-simulation equivalence of its next states. Checking for bi-simulation equivalence of two systems require that the entire state space of the two systems is available and known prior to equivalence checking which is challenging when behavior is not known and has to be computed dynamically or when we are dealing with infinite state space systems. Hence, we instead try to find an evidence to prove the non-bisimulation equivalence of two systems. This algorithm computes the next states and checks for equivalence only until the first evidence of non-bisimilarity is found. However, when the systems are indeed bisimilar, it explores the entire reachable state space. We also produce the evidence which is a path in the composed system behavior that supports the non-bisimulation result when the systems are not bi-simulation equivalent. In our tool, we use the notion of weak bi-simulation in checking the equivalence of two systems. Note that a possible downside in using bi-simulation checking in place of language equivalence is that systems that are language equivalent are not necessarily bi-simulation equivalent.

6. **Synchronizable/ Bounded System ($\mathcal{I}_k$)**

   If the two systems $\mathcal{I}_k$ and $\mathcal{I}_{k+1}$ are bi-simulation equivalent, then $\mathcal{I}$ is either synchronizable or k-bounded based on whether k is 0 or a value greater than 0 respectively. System $\mathcal{I}_k$ with bound of k has the behavior and language sequence that is similar to an unbounded version of $\mathcal{I}_k$.

7. **Local State DFS** This module takes as input a local state and a message queue to determine if all the messages in the queue can be consumed with out getting blocked. The

algorithm operates in a dfs-mechanism by moving to a next state looking for a transition that by doing zero or more sends, can consume the message at the head of the queue.

8. **Well-formedness Verifier** The well-formedness verifier module checks at each configuration of the composed system behavior, whether all the messages available in the message queues of each peer can be consumed. The algorithm marks all the configurations that are visited to avoid duplicate or redundant computations. As a configuration is composed of local states from each peer, the algorithm uses the Local State DFS module to verify for each local state in its current configuration whether all the messages in its respective queue at its current state can be consumed. If a configuration contains atleast one local state that cannot consume its messages in any of its paths, then the algorithm returns false indicating that the system is not well-formed.

9. **Composed System Behavior** If the system is synchronizable or k-bounded, we build the composed system behavior which is a list of configuration and transition objects that consists of the interactions between the configurations of the composite system.

10. **Visualization Graph** The Visualization Graph is a way of structuring the state configuration and transition objects into a format that can be displayed as diagrams of abstract graphs and networks.

11. **Output Modeler** The Output modeler takes the composed system behavior and converts into the format that can be used to generate the Visualization graph.

6.2   Experimental Results

Table 6.1 presents the experimental results for systems, that are synchronizable or k-send-bounded. We have used case studies from the existing literature that ranges from service contracts (Reservationsession, Metaconversation, etc.) to Singularity OS contracts (TpmContract, TcpContract, KeyboardContract).

For each example, the second column shows the number of peers participating in the interaction. The third column shows the total number of states and transitions, and the value
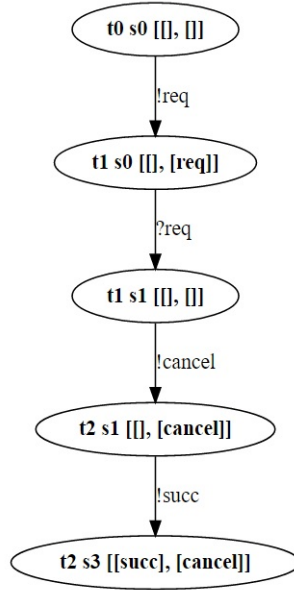
Figure 6.3   Trace of reservation session protocol : Not well-formed

of k for k-send-boundedness of the system. For the synchronizable systems, the corresponding k value is 0. The fourth column indicates our finding—whether or not the system is well-formed. The last column shows the time taken to find the k and the time for testing well-formedness.

For each system, we have verified either synchronizability or the k-send-boundedness conditions to identify the bound. Once such a bound is obtained, we verified for well-formed behavior for systems that are not synchronizable. Note that for synchronizable systems, well-formedness is guaranteed; therefore, we do not report time for verifying well-formedness. Figure 6.4 shows

Table 6.1   Results: Well-formedness

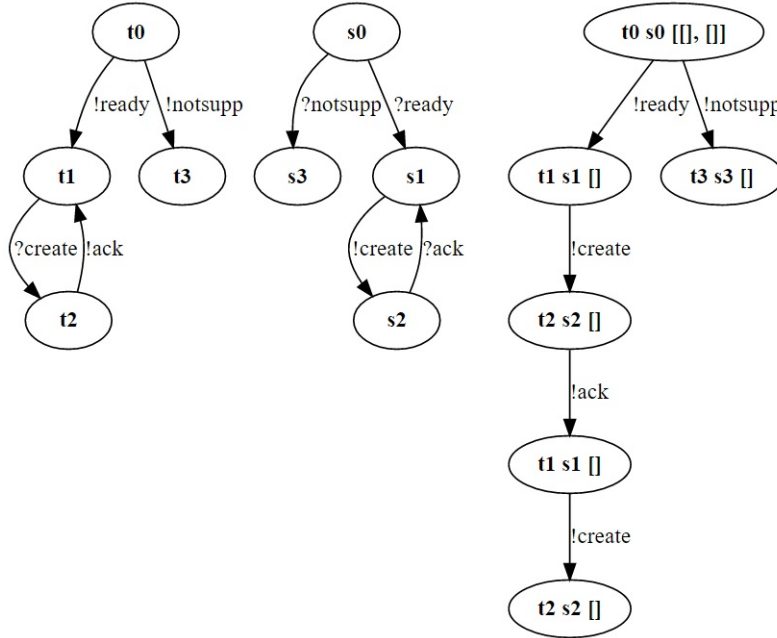| Case study	Peers	S/T/k	Well-Formed	Time (sec)
ReservationSession	2	12/16/1	✗	0.012/0.005
Metaconversation	2	8/12/1	✗	0.038/0.004
TpmContract	2	10/14/2	✗	0.016/0.004
TcpContract	2	8/8/0	✓	0.020/-
KeyboardContract	2	8/14/0	✓	0.036/-
News server Ouederni et al. (2014)	2	9/9/3	✗	0.020/0.005

Figure 6.4   tcp protocol: Synchronizable and well-formed

the tcp contract system with two peers. The tcp contract system is synchronizable and the figure shows the two peers along with its composite state behavior.

For the systems, that are not well-formed, we can automatically identify the path leading to the configuration from where at least one peer cannot consume the messages present in its queue. As an example, the reservationsession protocol is not well-formed. Figure 6.3 provides a path trace for the reservationsession protocol resulting from our algorithm.

From figure 6.3, the configuration $t_2s_3[[succ],[cancel]]$ is not well-formed as there is no transition out of state t_2 in zero or more send actions that can consume the message *succ* in its current configuration. For two-peer systems, well-formedness is violated when the peers move by sending each other messages and are not capable of consuming the messages sent to them.

We have also experimented with some Web services with more than two peers such as the Running Example in Akroun et al. (2016) and the Client-Supplier example in Happe et al. (2010). We converted the peers in these systems to form a network of communicating systems where every peer sends messages to only one other peer in a ring topology. That is, with respect

to each peer all other peers in the ring can be viewed as one peer, hence forming a two-peer asynchronous system. The example in Akroun et al. (2016) has a queue-bound of 1 and we have verified it to be well-formed. On the other hand, the example from Happe et al. (2010) is synchronizable and therefore, well-formed.

CHAPTER 7. CONCLUSION

7.1 Summary

In this work, our focus is on verification of well-formedness properties in asynchronous message-passing systems. Well-formedness property states that every message that is produced is eventually consumed. When peers in an asynchronous system communicate over their buffers, it is not always guaranteed that a sent message is eventually consumed which can happen when the receiving peer was never ready to receive the message leading to an unwanted behavior or a deadlocked state. Hence, verification of well-formedness for asynchronous communications is very important to ensure safe behavior in its evolution. However, verification of well-formedness requires that at every configuration in the state space, every local peer will be able to consume the messages in its buffer eventually. When the buffers are unbounded, computation of reachable state space is undecidable as discussed in Brand and Zafiropulo (1983), which makes the well-formedness verification problem challenging.

We prove in this thesis that verifying well-formedness in asynchronous systems with unbounded buffers is undecidable. The proof relies on simulation of Turing machine by asynchronously communicating finite state machines and reduction of the Halting problem for Turing machines to well-formedness verification problem in asynchronous systems. We also identify two important sub-classes of asynchronous systems for which verification of well-formedness can be automatically tested. These two sub-classes are the synchronizable subclass and the k-bounded subclass. Systems are synchronizable if and only if the interactions resulting from asynchronous communication over unbounded queues can be captured by the interactions resulting from synchronous communication where every sent-message is immediately consumed. We proved that synchronizable systems are implicitly well-formed. Another subclass is referred

to as k-send-bounded. Systems are k-send-bounded if and only if the interactions resulting from asynchronous communication over unbounded queues can be captured by the interactions resulting from asynchronous communication over k-bounded queues. We prove that for systems in k-send-bounded subclass, well-formedness property can be automatically verified.

We have built a tool to conduct experiments on various types of asynchronous systems and verify their language equivalence and well-formedness properties. The tool takes as input the asynchronous system represented as ptrans relations, translates into internal objects and verifies the synchronizability or the k-send-boundedness conditions to identify a bound on the system. If a bound is found, the tool verifies the well-formedness properties of the system. It also provides a trace path of system behavior as a witness when it finds the system is not well-formed. The tool can also be easily extended to work with different input representations of the asynchronous specifications and also to add new modules to verify behavior under varied conditions. We presented the results of verifying well-formedness properties for various case studies. Our experiments show that systems in different domains fall within these sub-classes, thus making automatic verification of well-formed behavior possible.

7.2   Future Work

We have proved that the well-formedness property is undecidable for message-passing asynchronous systems that communicate by FIFO queue. One of the primary areas that we are looking to focus is the verification of well-formedness property for different types of asynchronous systems. A primary assumption in the model of systems that we considered is that the ordering of the messages in the queue is first-in-first-out, where every peer in the composite system is assumed to have a buffer to manage the incoming messages. However, there are other variations of systems where in there is a channel for every sender-receiver pair as in Peer to Peer network protocols. Another variation is a system in which all the peers share a single channel, all sending peers send messages to the channel and receiving peers consume the messages from the channel. In both the above types, verifying well-formedness is likely to remain undecidable as the behavior is still equivalent in terms of message consumption. However, when we deal with systems where messages from a channel can be consumed in any order or allow for ignoring messages or

deferring message consumption (for example, systems with huge network transmission delays), the well-formedness property may not remain undecidable. We want to investigate more on different types of these systems and prove the verifiability of well-formedness in these systems.

Another area of our interest is to extend the work in the verification of well-formedness properties for domain specific languages. We currently have built an interpreter that translates a subset of Microsoft's P language semantics into our system representation and verify the well-formedness properties for certain programs in P. We would like to extend our interpreter to handle a broader set of semantics from the language that add interesting behaviors to the composite system; model the behavior and verify well-formedness properties for wider range of programs.

BIBLIOGRAPHY

Akroun, L., Salaün, G., and Ye, L. (2016). Automated analysis of asynchronously communicating systems. In *International Symposium on Model Checking Software*, pages 1–18. Springer International Publishing.

Armstrong, J. (2002). Getting Erlang to talk to the outside world. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 64–72.

Basu, S. and Bultan, T. (2011). Choreography conformance via synchronizability. In *20th International World Wide Web Conference*, pages 795–804.

Basu, S. and Bultan, T. (2014). Automatic verification of interactions in asynchronous systems with unbounded buffers. In *ACM/IEEE International Conference on Automated Software Engineering*, pages 743–754.

Bocchi, L., Chen, T., Demangeon, R., Honda, K., and Yoshida, N. (2013). Monitoring networks through multiparty session types. In *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE*, pages 50–65.

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

Cécé, G. and Finkel, A. (2005). Verification of programs with half-duplex communication. *Information and Computation*, 202:166–190.

Clarke, Jr., E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. MIT Press, Cambridge, MA, USA.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition.

Deniélou, P.-M. and Yoshida, N. (2012). Multiparty session types meet communicating automata. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 194–213, Berlin, Heidelberg. Springer-Verlag.

Desai, A., Gupta, V., Jackson, E. K., Qadeer, S., Rajamani, S. K., and Zufferey, D. (2013). P: safe asynchronous event-driven programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 321–332.

Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G. C., Larus, J. R., and Levi, S. (2006). Language support for fast and reliable message-based communication in singularity os. In *Proc. 2006 EuroSys Conf.*, pages 177–190.

Finkel, A. and McKenzie, P. (1997). Verifying identical communicating processes is undecidable. *Theoretical Computer Science*, 174(1):217 – 230.

Fu, X., Bultan, T., and Su, J. (2004). Analysis of interacting BPEL web services. In *Proc. 13th Int. World Wide Web Conf.*, pages 621 – 630.

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

Gouda, M. G. and Chang, C.-K. (1986). Proving liveness for networks of communicating finite state machines. *ACM Transactions on Programming Languages and Systems*, 8(1). Sufficient conditions for three different types of state-based liveness and characterization of three different types of state-based fairness.

Gouda, M. G., Gurari, E. M., Lai, T. H., and Rosier, L. E. (1987). On deadlock detection in systems of communcating finite state machines. *Computers and Artificial Intelligence*, 6(3):209–228.

Gouda, M. G., Manning, E. G., and Yu, Y. T. (1984). On the progress of communication between two finite state machines. *Information and Control*, 63:200–216.

Happe, J., Buhnova, B., Cmara, J., Martn, J. A., Salaün, G., Canal, C., and Pimentel, E. (2010). Semi-automatic specification of behavioural service adaptation contracts. *Electronic Notes in Theoretical Computer Science*, 264(1):19 – 34.

Honda, K., Yoshida, N., and Carbone, M. (2008). Multiparty Asynchronous Session Types. In *Proceedings of Symposium Principles of Programming Languages*, pages 273–284.

Manohar, R. and Martin, A. J. (1998). Slack elasticity in concurrent computing. In *Mathematics of Program Construction, (MPC)*, pages 272–285.

Ouederni, M., Salaün, G., and Bultan, T. (2014). Compatibility checking for asynchronously communicating software. In *Formal Aspects of Component Software: 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*, pages 310–328. Springer International Publishing.

Siegel, S. F. (2005). Efficient verification of halting properties for MPI programs with wildcard receives. In *6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 413–429.

Singularity (2004). Singularity design note 5 : Channel contracts. singularity rdk documentation (v1.1). http://www.codeplex.com/singularity.

Vakkalanka, S., Vo, A., Gopalakrishnan, G., and Kirby, R. M. (2010). Precise dynamic analysis for slack elasticity: adding buffering without adding bugs. In *17th Euro. MPI Conf. Advances in Message Passing Interface*, pages 152–159.

West, C. H. (1978). General technique for communications protocol validation. *IBM J. Res. Dev.*, 22(4):393–404.

WS-CDL (2006). Web Service Choregraphy Description Language. https://www.w3.org/TR/ws-cdl-10-primer/.

Yu, Y. T. and Gouda, M. G. (1982). Deadlock detection for a class of communicating finite state machines. *IEEE Transactions on Communications*, COM-30(12):2514–2518.