

# Automated Verification of Proper Choreography Implementation

Erwin Jansen, Hen-I Yang and Sumi Helal  
Dept. of Computer & Information Science & Engineering  
University of Florida Gainesville, FL-32611  
{ejansen,hyang,helal}@cise.ufl.edu

## Abstract

*As business transactions become more complex it is important that all participants obey the so called rules of engagement. These rules of engagement can be captured in an interaction protocol or choreography. A choreography provides rules on how various components act together, in terms of observable behavior. We investigate how we can extract a protocol from a program and verify whether or not the implementation adheres to the protocol. We derive an algorithm to verify whether a program is operationally compatible with a protocol, meaning the program does not violate the protocol. This guarantees that a program will not enter a dead lock or will fail due to unexpected messages.*

Keywords: Internet workflow, Cooperation, Infrastructure for e-services, Brokering, Webservice, Smart home

## 1. Introduction

Agents interact with each other in order to exchange information. When two agents are exchanging messages with each other we will say that two agents are engaged in a conversation. Agents that are engaged in a conversation must adhere to a so called protocol of interaction. A protocol of interaction defines which messages an agent understands and which messages an agent can send to, or expect from another agent.

Consider the example of a monotonic bargaining protocol. The protocol specifies how two parties can come to an agreement on the sale of a particular item. Figure 1 shows this protocol.

Once proper protocols are defined, it is crucial to verify whether the implementation of agents adhere to them. If an agent has an improper implementation of a protocol, its conversations can cause mishandling, unpredictable behaviors or even halt the interaction when engaged in a conversation.

Intuitively, one would be tempted to verify the language equivalence between automata defined by the protocol and

the implementation of an agent, since equivalence guarantees the same language would be accepted hence operating in similar fashions. This inclination, however, proves to be insufficient for verifying whether implementations are legitimate. Therefore, a more relaxed criterion has to be established, and an algorithm with reasonable complexity has to be created for the verification purpose.

Imagine a software agent representing the purchasing department of a company. Using web services, the agent sifted through potential manufacturers on the Internet, and starts an automatic negotiation process.

The bargaining protocol dictates possible states, acceptable messages associated with each state, and the synchronized transitions between states for each participant. If the agent representing the purchasing department is certified to participate as buyer of the bargaining protocol, and the web service representing the manufacturer is certified to participate as seller of the same protocol, bargaining would take place automatically, and eventually either an agreement would be reached or the negotiation would be aborted. To avoid any fallout in the bargaining process, the purchasing agent should choose only to negotiate with those web services that are certified for the seller role of the bargaining protocol.

The value of protocols and verification of agents against them is that it guarantees that during the negotiation process there would be no failure caused by the conversation. Each participant knows how to react to any message it receives from its counterparts. Instead of interfacing proprietary software systems, web services along with protocols ensure smooth exchange of longer and more complex multi-stage conversations. With the algorithm presented in this paper, automatic verification of agent implementations against any protocol becomes feasible, hence guarantees all participants would play by the rule.

Another use of the automatic agent verification against protocols can be found in a self-integrating smart house. The interoperability issue caused by introduction of new devices into existing smart environment is a well-known problem in pervasive computing research. We are investigating

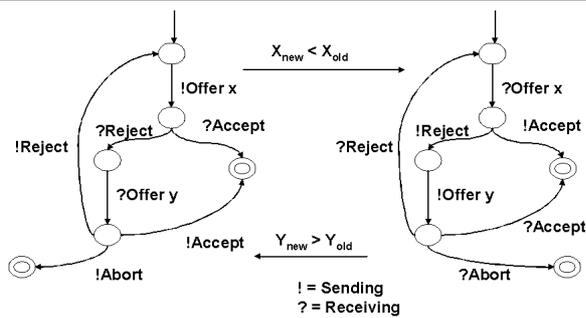


Figure 1. The monotonic bargaining protocol.

the use of automatic verification algorithm as a safe guarding measure to disallow service bundles that fail to be certified for certain protocols from being introduced to the runtime environment of the Gator Tech Smart House (GTSH [2]) as to prevent conflicting or confusing interactions between the new service bundle and the existing ones.

Consider a service that provides mobile TV on cellular phone. The cellular phone has limited processing capabilities. Now suppose that when the phone is introduced to the GTSH it wants to surrogate some of the video rendering to the desktop computer already in the house. Two service bundles representing the cellular phone and the computer respectively would embark on a bargaining process to decide how much processing power the computer can provide. In order to make sure the automatic bargaining can progress smoothly, it is essential that we make sure the bundle which represents cellular phone adheres to the bargaining protocol. If it is not certified to bargain as buyer, the smart house runtime environment should ban the surrogating of rendering from even happen at all. The last thing that we want to see happen is that the whole smart house goes crazy because the new sleek cellular phone ask for help on video rendering.

## 2. Protocol of Interaction

There are two aspects that are relevant to a protocol of interaction:

**Static Component:** This defines the flow of interaction. Agents exchanges messages over channels. The static component defines what *type* of messages are allowed at any given time during the execution of the protocol. For example it specifies that after an offer has been transmitted the opposing party can either send an accept or reject message.

**Dynamic Component:** This defines which constraints must be met by the values, of the type defined in the

static component, that are exchanged over the channels. In the bargaining protocol for example we have the constraint that a buyer can only ask a lower price than the price requested earlier.

### 2.1. Static Component

Protocols can be expressed using communicating finite state machines (CFSM[1]), a finite state machine that indicates what messages an agent can receive or send in the current state. A *well formed* protocol makes sure that agents never enter a state such that one agent sends a messages that cannot be understood by the other.

When an agent system consists of multiple agents that interact with each other we want to be certain the protocol is well formed and does not contain any deadlocks. This would assure that the agent system does not encounter execution failures due to failed communication. From a stable protocol we expect the following two things:

**Well formed:** A protocol has a missing reception if a message  $x$  can arrive at an agent when it is in state  $s$ , but the protocol does not specify which state should be entered upon receiving  $x$  in state  $s$ . A protocol without missing receptions is well formed.

**Deadlock:** A protocol is in a deadlock when both agents are either waiting to send a message, or both agents are waiting to receive a message.

There are automated techniques to verify if a protocol is well formed and does not contain any deadlocks. These techniques are based upon model checking and suffer from the limitation that a model of the protocol must be extracted from an entire agent system. A state machine is generated for every individual agent. These state machines are later combined into a global state machine, called a product machine. For every state in the machine we then verify whether or not the state is stable (i.e. violates any of the two properties mentioned above). Model checking techniques based upon reachability analysis are expensive and not always feasible [4, 9, 8].

Extracting a protocol from a set of webservices is unrealistic and not always necessary. Most business will specify the protocols of interaction and make these publicly available. E-bay could for example make a bargaining protocol available that specifies how a buyer can buy a good from a seller.

Given that we have the protocol we must make sure that an agent implements this protocol properly. That is, can the agent act as a participant and not destabilize the protocol?

## 2.2. Dynamic Component

The dynamic component of a protocol defines the conditions that must be met by the data that is actually transferred over the channels. The dynamic part specifies the contract between the sender and receiver. This part of the protocol specifies what the receiver expects of a sender in term of values that have been transmitted.

Enforcing the protocols of interaction relates to the pre- and postconditions as used by design by contract[5]. The difference is that design by contract implicitly assumes that the interactions between a client and server are atomic. There is no specification of conditions on more complex forms of interactions. However as the monotonic bargain protocol clearly demonstrates such a history does have an impact of which messages are allowed or not.

The dynamic part of the protocol verification has to be done by observing and approving the messages that are exchanged between participants. The entity responsible for the exchange of the messages must examine the transmitted messages and verify if one of the conditions are being broken. If a protocol failure arises an exception mechanism can take care of handling the failure.

For the remaining part of this paper, we focus on automated verification of agents against static component of stable protocols.

## 3. Formal Definition of Protocol of Interaction

Protocols can be defined as a collection of finite state machines that are able to send and receive messages from other state machines. Formally a protocol can be defined as follows:

**Definition 1 (Protocol[8]).** A protocol  $\Pi$  is a pair  $(P, L)$  where  $P = \{P_i \mid i \in I\}$  a set of  $n$  processes where  $I = \{1, 2, \dots, n\}$  a finite index set, with  $n \geq 2$ . Furthermore we have:

- Let  $L \subseteq I \times I$  be an irreflexive incidence relation set of error-free simplex channels  $\{C_{ij} \mid (i, j) \in L\}$ .
- Each  $P_i = (X_i, x_0, \tilde{M}_i, F_i, \delta_i)$  is a finite state machine that can communicate with the other processes using the channels.
- We define  $X_i$  to be a finite set of states of  $i$  and the initial state  $x_i^0 \in X_i$ .
- Let  $M_{ij}$  denote the set of message that  $P_i$  can send over channel  $C_{ij}$ .  $M_{ij} = \emptyset$  if  $(i, j) \notin L$ . Similarly we define  $M_{ji}$  the set of message that  $P_i$  can receive over channel  $C_{ji}$ .  $M_{ji} = \emptyset$  if  $(j, i) \notin L$ . We write  $!m$  when  $m \in M_{ij}$  and  $?m$  whenever  $m \in M_{ji}$ .

- Let  $\tilde{M}_i = \bigcup_{j \in I} \{m_{ij} \cdot x \mid x \in M_{ij}\} \cup \bigcup_{j \in I} \{m_{ji} \cdot x \mid x \in M_{ji}\}$ . Where  $a \cdot b$  denotes the concatenation of the string  $a$  and  $b$ .
- $\delta_i \subseteq X_i \times \tilde{M} \times X_i$  denote the transition relation for process  $P_i$ . We will write  $\delta_i(x) = \{(m, x') \mid (x, m, x') \in \delta_i\}$  for the transitions from  $x$  and  $\delta_i(x, m) = \{x' \mid (x, m, x') \in \delta_i\}$  for the destination of  $x$  after receiving/sending  $m$ . We will write  $\delta(x, \tau)$  for internal transitions that do not receive or sends any message. Note that this is different from sending the empty message  $\epsilon$ . We will also use  $x \xrightarrow{m} x'$  as another way of writing  $(x, m, x') \in \delta$ .
- $F_i \subseteq X_i$  denotes the set of accepting states.

We can detect errors in a protocol by generating a global state machine and check if every state that is reachable from the initial state is a stable state, that is it is not a deadlock state or a state with missing receptions. The global state machine is generated by creating a so called product machine of all the individual state machines:

**Definition 2 (Global State[8]).** Let  $\langle a_i \rangle_{i \in Z}$  denote a  $|Z|$ -tuple  $(a_{i_1}, a_{i_2}, \dots, a_{i_{|Z|}})$ . A global state  $G$  for a protocol  $\Pi$  is represented as a pair  $(\langle x_i^G \rangle_{i \in I}, \langle c_{ij}^G \rangle_{(i,j) \in L})$  where  $x_i^G$  is the local state of process  $P_i$  in  $G$  and  $c_{ij}^G$  the content of channel  $C_{ij}$ . In particular, the initial global state  $G^0 = (\langle x_i^{G^0} \rangle_{i \in I}, \langle c_{ij}^{G^0} \rangle_{(i,j) \in L})$  of  $\Pi$  is such that  $\forall i \in I \forall (i,j) \in L \langle x_i^{G^0} = x_i^0, c_{ij}^{G^0} = \epsilon \rangle$

The definition above applies to both synchronous and a-synchronous systems. In a synchronous system the channels are bound and do not contain any messages at all, since the message is delivered immediately.

From a global state we can reach other global states by making a global transition. Global transitions are defined in terms of local transitions, a single state machine makes a transition leading to a new global state. Since not every local transition can occur as a global transition, we define global transitions in terms of *executable transitions*. Executable transitions are transitions of a local state machine that can be executed in a global state.

The introduction of executable transitions makes the distinction between an a-synchronous and a synchronous system clear:

**A-Synchronous:** In an a-synchronous system the sending of a message will always succeed whereas the receiving of a message can only succeed if that particular message is the first message in the channel from which we are receiving the message.

**Synchronous** In a synchronous system the sending of a message can *only* succeed if it is matched with the receiving of the same message along a channel.

The main difference between the two is that a global state transition in a synchronous system consists of the transition of two individual state machines whereas in an a-synchronous system it consists of only a single transition. The verification of stability of a protocol is similar for both a-synchronous and synchronous systems.

**Definition 3 (Executable Transitions[8]).** For an a-synchronous system a transition  $t = (x_i, z, x'_i)$  of  $P_i$  is executable at a global state  $G$  iff  $x_i = x_i^G$  and one of the following holds:

1.  $t$  is a send transition such that  $z = m_{ij} \cdot y$  with  $y \in \bigcup_{j \in I} M_{ij}$  and  $c_{ij} = y$ .
2.  $t$  is a receive transition such that  $z = m_{ji} \cdot y$  with  $y \in \bigcup_{j \in I} M_{ji}$  and  $c_{ji} = y$ .

For a synchronous system two transitions  $t = (x_i, m_{ij} \cdot y, x'_i)$  of  $P_i$  and  $t' = (x_j, m_{ji} \cdot z, x'_j)$  of  $P_j$  are executable at a global state  $G$  iff  $x_i = x_i^G$  and  $x_j = x_j^G$  and  $z = y$ .

The sets of send transitions and receive transitions of  $P_i$  that are executable at  $G$  are denoted by  $T_i^-(G)$  and  $T_i^+(G)$  respectively.  $T_i(G) = T_i^-(G) \cup T_i^+(G)$ .  $T(G) = \bigcup_{i \in I} T_i(G)$ .

The set of executable transitions consists of all the possible transitions that are executable for each individual state machine. From the set of executable transitions in a given state we can define when a global transition can occur:

**Definition 4 (Global Transition).** We define a global transition  $G \rightarrow H$  if  $\exists i \in I$  such that  $H$  can be derived from  $G$  by executing a single transition  $t \in T_i(G)$  according to the following conditions:

1.  $t = (x_i^G, m_{ij} \cdot y, x_i^H)$  for some  $y \in \bigcup_{j \in I} M_{ij}$  and  $c_{ij}^H = c_{ij}^G \cdot y$
2.  $t = (x_i^G, m_{ji} \cdot y, x_i^H)$  for some  $y \in \bigcup_{j \in I} M_{ji}$  and  $c_{ji}^H = y \cdot c_{ji}^G$

Notice that  $\mathbf{P}_\Pi = \langle \mathbf{G}, \rightarrow, \bigcup_{i \in I} \tilde{M}_i \rangle$ , where  $\mathbf{G}$  is the set of all global states, is the product machine that is derived from the individual state machines defining  $\Pi$ .

Automata specify what each participant can do in a protocol. We say that a protocol is minimal if none of its participants has a transition in its local state machine that does not occur in the global state machine:

**Definition 5 (Minimal Protocol).** We call a protocol  $\Pi$  minimal if for every  $P_i \in \Pi$  and  $(x, z, x') \in \delta_i$  there is a  $G$  such that  $(x, z, x') \in T(G)$

Notice that such a minimal protocol always exists and can be derived easily. A protocol of interaction defines which messages can be exchanged over the channels of communication. Hence a protocol does never contain any

empty transitions and we can minimize a protocol using hopcroft's algorithm [3].

We now have all the base ingredients to verify whether a state machine implements a stable protocol.

## 4. Verifying Correct Implementation

In order for an agent to be a correct participant in a protocol we need to examine the concept of equivalence. If two automata are equivalent, we could replace one automaton with the other without introducing any unstable states in the product machine.

Thus if we have a protocol  $P$  we need to verify that agent  $A$  properly implements this protocol. That is we need to see if we can replace the automaton obtained from the program of  $A$  with any of the participants  $P_i \in P$ . We can obtain the automaton from a program by examining the flow graph of a program:

**Automaton Construction:** Each statement in the program is a node in the flow graph; if a statement  $x$  can be followed by statement  $y$ , there is a node from  $x$  to  $y$ . We can extend the flow graph by labelling the edges with  $\tau$  for statements that perform internal calculations and labels from  $\tilde{M}$  for communication actions.

One is tempted to verify the language equivalence between two automata and argue that since they both accept the same language they both will operate in a similar fashion. This is however not the case.

If we would consider language equivalence to be sufficient we would consider a deterministic system similar to a non-deterministic system<sup>1</sup>. However systems might have internal non-determinism that can cause them to operate very differently then their deterministic counterpart [6]!

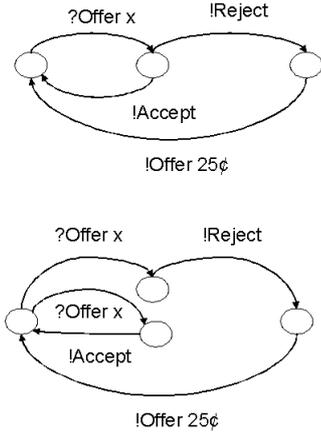
In figure 2 for example we have two automata that are both acting as participants of the monotonic bargaining protocol. Both automata produce the same language, yet the way they operate is entirely different. One automaton makes its decision without even looking at the message, whereas the other possibly investigates if the offer was any good to begin with.

## 5. Strong Equivalence

As we argued earlier on language equivalence does not capture the notion of internal non-determinism and is not strong enough to be used to identify whether an automaton  $Q$  can act as a participant  $P_i$  in a stable protocol.

The reason that language equivalence is not strong enough stems from the fact that language equivalence

<sup>1</sup> A non-deterministic automaton can always be converted into a deterministic one.



**Figure 2. Language Equivalent Bargainers**

does not identify how a certain automaton came to accepting a string. It merely defines that an automaton is capable of accepting a string. Not only do we need to capture language equivalence, but we must also keep track of the states that we encountered to accept the string.

We therefore introduce the concept of strong simulation. An automaton  $P$  is strongly simulating  $Q$  if the behavior of automaton  $P$  is rich enough to 'simulate' the behavior of automaton  $Q$ :

**Definition 6 (Strong Simulation).** Let  $P = (X_P, \tilde{M}, \delta_P), Q = (X_Q, \tilde{M}, \delta_Q)$  be two automata. Let  $\mathcal{S} \subseteq X_P \times X_Q$  be a binary relation over the set of states. Then  $\mathcal{S}$  is said to be a strong simulation if, whenever  $x_P \mathcal{S} x_Q$ ,

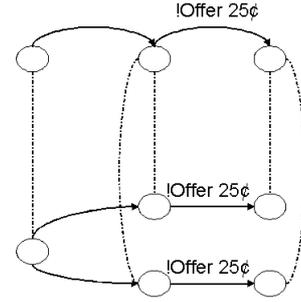
if  $x_P \xrightarrow{m} x'_P$  then there exists  $x'_Q \in X_Q$  such that  $x_Q \xrightarrow{m} x'_Q$  and  $x'_P \mathcal{S} x'_Q$

We say that  $x_Q$  strongly simulates  $x_P$  if there exists a strong simulation  $\mathcal{S}$  such that  $x_P \mathcal{S} x_Q$ .

Based upon the notion of strong simulation we can define a form of equivalence called strong equivalence. An automaton  $P$  is strongly equivalent to an automaton  $Q$  if both automata are able to simulate each other:

**Definition 7 (Strong Bisimulation and Equivalence).** A binary relation  $\mathcal{S} \subseteq X_P \times X_Q$  is said to be a strong bisimulation if both  $\mathcal{S}$  and its converse<sup>2</sup>  $\mathcal{S}^{-1}$  are strong simulations. We say that  $P$  and  $Q$  are strongly bisimilar or strongly equivalent written  $P \sim Q$  if there exists a strong bisimulation  $\mathcal{S}$  such that  $PSQ$

In figure 3 we give an example of two automata that are strongly equivalent.



**Figure 3. Strong Equivalent Bargainers**

Strong equivalence gives us a first notion on when we can replace an automaton in a protocol without violating its stability. Since two strongly equivalent automata behave the same, we can replace a participant in a protocol with a strongly equivalent automaton. The protocol where we replaced the participant will still be stable:

**Theorem 1.** If  $P \sim Q$  and the protocol  $\Pi$  in which  $P$  participates is stable, then the protocol obtained from replacing  $P$  with  $Q$  is stable as well.

*Proof.* For every  $x_P \xrightarrow{m} x'_P$  there exists a  $y_Q \xrightarrow{m} y'_Q$ . Since  $x_P \xrightarrow{m} x'_P$  leads to a stable state, so does  $y_Q \xrightarrow{m} y'_Q$ . We also have that for every  $y_Q \xrightarrow{m} y'_Q$  there exists a  $x_P \xrightarrow{m} x'_P$ , hence  $Q$  does not have any transitions that  $P$  does not have.  $\square$

## 6. Observational Equivalence

Although strong equivalence gives us a good start on deciding whether an automaton  $Q$  can act as a participant  $P_i$  in a protocol it might be too strong. Up until now we conveniently left the internal transitions out of consideration. An internal transition is a computation that is not observable from an outsider. Consider the machines depicted in figure 4. It is clear that they are not strongly equivalent. However they seem to act in a similar fashion. Both send a counter offer, but one bargainer makes some internal computation before sending the counter offer.

The reason for this is that any action within the agent system that is internal cannot be observed by any other agent except the one executing this action. The communication actions that make use of a communication channel are however *observable* either by another agent or by an external observer who has access to the set of channels.

If an observer would examine the communication channels it would not notice the difference between an agent that performs an internal transition and then sends a message and an agent that directly sends a message. In order to make this explicit we define an observer relation:

<sup>2</sup> The converse of any binary relation  $R$  is the set of pairs  $(x, y)$  such that  $(y, x) \in R$

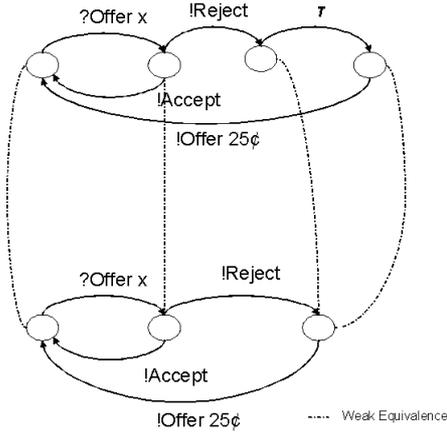


Figure 4. Weakly Equivalent Bargainers

**Definition 8 (Observer Relation).** The relation  $\Delta \subseteq X \times \tilde{M}^* \times X$ , where  $\tilde{M}^* = \{\tau\} \cup \tilde{M} \cup \tilde{M} \cdot \tilde{M} \cup \tilde{M} \cdot \tilde{M} \cdot \tilde{M} \cup \dots$ . Instead of  $(x, s, x') \in \Delta$  we will write  $x \Rightarrow x'$ .  $\Rightarrow$  is defined as follows:

1.  $x \Rightarrow y$  means there is a sequence of zero or more transitions  $x \xrightarrow{\tau} x_1 \xrightarrow{\tau} \dots \xrightarrow{\tau} x_n \xrightarrow{\tau} y$ . Formally  $\Rightarrow = \Rightarrow^*$  the transitive reflexive closure of  $\rightarrow$ .
2. Let  $s = m_1 \dots m_n$  then  $x \xRightarrow{s} y$  means  $x \Rightarrow x_1 \xrightarrow{m_1} x_2 \dots \Rightarrow x_n \xrightarrow{m_n} x'_n \Rightarrow y$ . Formally  $\Rightarrow \xRightarrow{s} \Rightarrow \xrightarrow{m_1} \Rightarrow \dots \Rightarrow \xrightarrow{m_n} \Rightarrow$

Now  $\xRightarrow{m}$  is a message that has been communicated, accompanied by a number of internal actions that either occur before or after the exchange of  $m$ .  $\xRightarrow{\tau}$  denotes one or more internal computations, whereas  $\Rightarrow$  denotes zero or more internal computations. Using the observer relation we can define weak equivalence, which allows us to identify automata as similar even if one is performing more internal transitions than the other:

**Definition 9 (Weak Simulation).** Let  $P = (X_P, \tilde{M}, \delta_P), Q = (X_Q, \tilde{M}, \delta_Q)$  be two automata. Let  $\mathcal{S} \subseteq X_P \times X_Q$  be a binary relation over the set of states. Let  $m$  be a sequence of actions. Then  $\mathcal{S}$  is said to be a weak simulation if, whenever  $x_P \mathcal{S} x_Q$ ,

if  $x_P \xRightarrow{m} x'_P$  then there exists  $x'_Q \in X_Q$  such that  $x_Q \xRightarrow{m} x'_Q$  and  $x'_P \mathcal{S} x'_Q$

We say that  $x_Q$  weakly simulates  $x_P$  if there exists a weak simulation  $\mathcal{S}$  such that  $x_P \mathcal{S} x_Q$

Verifying weak equivalence can become rather complicated since we have to check for all pairs  $(x_p, x_q) \in \mathcal{S}$  whether  $Q$  can match  $P$  for all observable actions. However as shown in [6] we only have to verify whether the following holds:

**Theorem 2.**  $\mathcal{S}$  is a weak simulation if and only if, whenever  $x_P \mathcal{S} x_Q$ :

$x_P \rightarrow x'_P$  then there exists a  $x'_Q \in X_Q$  such that  $x_Q \Rightarrow x'_Q$  and  $x'_P \mathcal{S} x'_Q$

$x_P \xrightarrow{m} x'_P$  then there exists a  $x'_Q \in X_Q$  such that  $x_Q \xRightarrow{m} x'_Q$  and  $x'_P \mathcal{S} x'_Q$

We now have all the ingredients to define weak bisimulation. The definition of weak bisimulation is the same as that of a strong bisimulation, we just replace  $\rightarrow$  with  $\Rightarrow$ :

**Definition 10 (Weak Bisimulation and Equivalence).** A binary relation  $\mathcal{S} \subseteq X_P \times X_Q$  is said to be a weak bisimulation if both  $\mathcal{S}$  and its converse  $\mathcal{S}^{-1}$  are weak simulations. We say that  $P$  and  $Q$  are weakly bisimilar or weakly equivalent written  $P \approx Q$  if there exists a weak bisimulation  $\mathcal{S}$  such that  $P \mathcal{S} Q$

An automaton that is weakly equivalent to a participant in a protocol can act as this participant without creating any unstable states. Since they act the same with respect to the actions that other automata can observe:

**Theorem 3.** If  $P \approx Q$  and the protocol  $\Pi$  in which  $P$  participates is stable, then the protocol obtained from replacing  $P$  with  $Q$  is stable as well.

*Proof.* For every  $x_P \xRightarrow{m} x'_P$  there exists a  $y_Q \xRightarrow{m} y'_Q$ . Since  $x_P \xRightarrow{m} x'_P$  leads to a stable state, so does  $y_Q \xRightarrow{m} y'_Q$ . We also have that for every  $y_Q \xRightarrow{m} y'_Q$  there exists a  $x_P \xRightarrow{m} x'_P$ , hence  $Q$  does not have any transitions that  $P$  does not have.  $\square$

## 7. Operation Compatibility

In the previous section we investigated under which conditions we can replace automata without destabilizing the protocol. We investigated equivalence relations that determine when two automata are compatible. We saw that equivalent automata can be interchanged without violating the stability property of a protocol.

A Protocol specifies which actions are valid and which actions are not valid. Consider an agent that implements the bargaining protocol but will never abort a negotiation and will always try to reach an agreement. This agent can be implemented by leaving out the abort transition in the bargaining protocol. Clearly this does not break the protocol, yet the automaton obtained from this agent is not weakly equivalent with any participants of the bargaining protocol. In this sense the concept of weak equivalence is still too strong.

An automaton  $Q$  could weakly simulate only a subset of another automaton  $P$  and act as participant  $P_i$  in a stable protocol without introducing any unstable states. We will

investigate under which conditions an automaton that is not weakly equivalent can act as participant in a protocol without creating unstable states.

## 7.1. Options in Protocols

In a synchronous system every communication action of an agent must be matched with another agent. However in order for a transition to occur one of the participants in the protocol must make a decision on which transition to take. At this point a subtle difference between an a-synchronous and synchronous system arises.

The distinction, from a formal perspective, between a synchronous and a-synchronous system has to do with the fact that in a synchronous system there is no difference between the sending and the receiving of messages:

**A-Synchronous:** If a sender has a choice of two or more possible messages that it can send it can make a decision independent of the receiver and proceed.

**Synchronous:** If a sender has a choice of two or more possible messages that it can send it can *only* make a decision if the receiver makes the same decision.

For the sake of simplicity we will only consider a-synchronous systems with a channel bound of one, meaning that a channel can only contain one message and the sender has to wait till the receiver has consumed this message. First of all this restriction is applicable for most of the system we can find in the real world and secondly the theory can be extended to deal with the intricacies of larger channels and synchronous systems.

We can now formally define a decision as follows:

**Definition 11 (An Option).** We say that an agent  $i$  has an option, whenever the following holds:

1.  $\Delta(x, !m) > 1$ , where  $m$  is an arbitrary message that is to be send.
2.  $\delta(x, ?m) \geq 1$ , where  $m$  is an arbitrary message that can be received.

We call the first one a send option and the second one a receive option.

## 7.2. Criteria for Operational Compatibility

The following theorem states that in any given state we either only send or only receive messages. It states that the mixing of send and receive transitions could potentially lead to a dead-lock.

**Theorem 4 (Exclusive Sending or Receiving).** Let  $P_i$  be a participant in a stable protocol. We have  $\forall s \in P_i \forall m \neg \exists z ((s \xrightarrow{?m} s_1 \wedge s \xrightarrow{!z} s_2))$ .

*Proof.* Suppose that an agent  $A$  has reachable states from  $s$  that are not uniform, i.e. there is a state  $s_1$  that can send message  $m$  and  $s_2$  that can receive a message  $n$ . Since the protocol is stable this means that there is another agent  $B$  that has can receive this message  $m$  as well as send a message  $n$ , but this implies that  $A$  can choose to send  $m$  and  $B$  can choose to send  $n$ . If both are sending at the same time this would lead to a deadlock, a contradiction with the stability property of the protocol.  $\square$

The following theorem states that we can add a receive transition of to any state of  $A$  that already contains a receive transition, without violating the protocol.

**Theorem 5 (Growing Receiving Options).** Let  $P_i$  be an automaton in a stable protocol. For any automaton  $A \approx P_i$  and any state  $s \in A$  such that there exists a  $s \xrightarrow{?m} s_1$  we can add a receiving option  $n \neq m$  such that  $s \xrightarrow{?n} s_2$  without violating the protocol.

*Proof.* This is trivial, since the protocol of  $A$  was stable already, adding extra messages that the other will not send will never be taken.  $\square$

Similarly to the previous theorem we can state that we can remove send options if we have more than one option we can choose from. By removing options we still have a stable protocol.

**Theorem 6 (Pruning Sending Options).** Let  $P_i$  be an automaton in a stable protocol. For any automaton  $A \approx P_i$  we can remove any send option without violating the protocol.

*Proof.* Trivial, in a stable protocol every message you send leads to a stable state. Hence if there are send options we have more than one transitions resulting in the sending of a message. If we remove one of these transitions the other transition(s) will still lead to a stable state.  $\square$

The transitions that we can remove are the send options that basically decide which communication action an agent will undertake. Each path through the state machine of a participant is stable, so always taking the same path just makes the decisions static, rather than dynamic. As long as we obey theorems 4, 5 and theorem 6 we should be safe.

## 8. Operational Compatibility Verification Algorithm

We are now ready to present an algorithm that simply verifies that we do not violate any of the previous stated theorems. The algorithm is straightforward, we go over all the states of the implementation machine and verify that there

is a mapping from each state to a similar state in the protocol that does not violate the growing, shrinking and exclusivity theorem.

The TClosure algorithm calculates the transitive closure of a state of the automaton and verifies that this state satisfies the exclusivity theorem. From the closure of the state we can either send or receive messages but not both.

```

TCLOSURE( $s$ )
1  if  $A[s] \neq \emptyset$ 
2    then return  $A[s]$ 
3   $A[s] \leftarrow \emptyset$ ;
4  for each  $s'$  in  $\delta(s)$ 
5    do if  $(s, \tau, s')$ 
6      then  $A[s] \leftarrow A[s] \cup \text{TCLOSURE}(s')$ 
7      else if  $A[s].type = m.type$ 
8        then  $A[s] \leftarrow A[s] \cup \{(s, m, s')\}$ 
9        else error "Violate exclusivity"
10 return  $A[s]$ 

```

From line 9-13 in the implements algorithm we verify if the state we can reach from  $q$  violates the pruning sending theorem, if it doesn't we add all reachable states from the closure to the ones to be verified. Similarly we verify the growing receiving theorem in line 14-18.

```

IMPLEMENTS( $q, p$ )
1   $L \leftarrow (s_0^p, s_0^q)$ 
2   $R \leftarrow \emptyset$ 
3  while  $L \neq \emptyset$ 
4    do  $(s_p, s_q) \leftarrow \text{head}(L)$ 
5     $R \leftarrow R \cup (s_p, s_q)$ 
6     $Q \leftarrow \text{TCLOSURE}(s_q)$ 
7     $P \leftarrow \text{TCLOSURE}(s_p)$ 
8    if  $Q.type = \text{send}$ 
9      then for each  $(s_q, m, s'_q)$  in  $Q$ 
10     do if  $\exists (s_p, m, s'_p) \in P$ 
11       then if  $(s'_p, s'_q) \notin R$ 
12         then  $L \leftarrow L \cup (s'_p, s'_q)$ 
13         else error "violated send"
14     else for each  $(s_p, m, s'_p)$  in  $P$ 
15       do if  $\exists (s_q, m, s'_q) \in Q$ 
16         then if  $(s'_p, s'_q) \notin R$ 
17           then  $L \leftarrow L \cup (s'_p, s'_q)$ 
18           else error "violated receive"

```

**Theorem 7.** *The above algorithm is correct.*

*Proof.* The algorithm simply verifies whether or not we violate theorem 4, 5 and 6.  $\square$

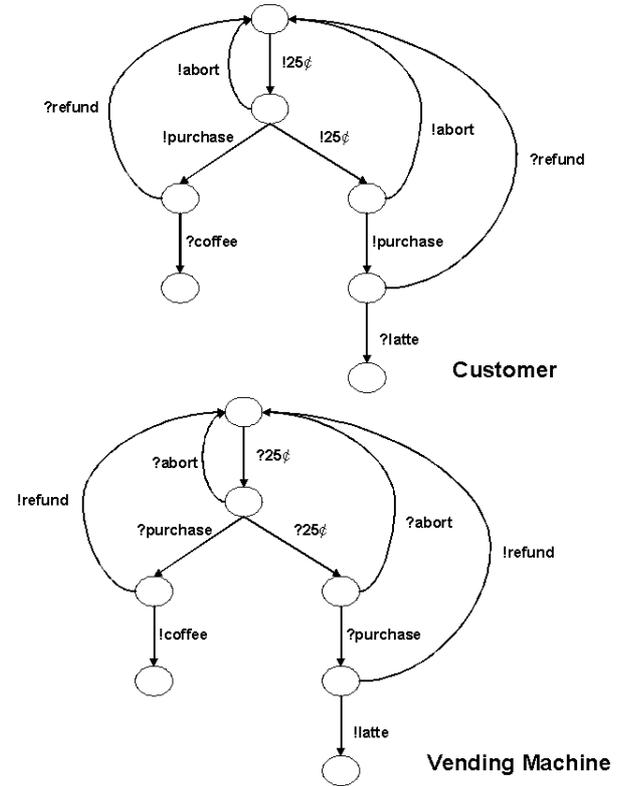
**Theorem 8.** *The above algorithm runs in  $\Theta(V^2E)$ .*

*Proof.* Calculating the transitive closure is a depth first search, and thus takes  $O(V + E)$ . The main contribution to the complexity is the fact that in the worst case every state

of  $P$  has to be mapped to every state of  $Q$ . Leading to  $V^2$  elements that are placed in  $L$ . Since we have to consider every edge in line 9 or 14 from that particular state we end up with  $\Theta(V^2E)$ .  $\square$

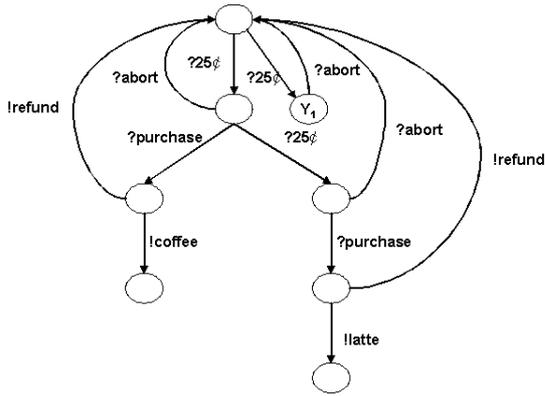
## 9. Example

In the section we'll use the "Coffee Vending Machine" protocol, defined for a simplified coffee vending machine, loosely based on [6], to illustrate the desired relation that verifies whether an implementation is capable to act as a participant in the protocol.



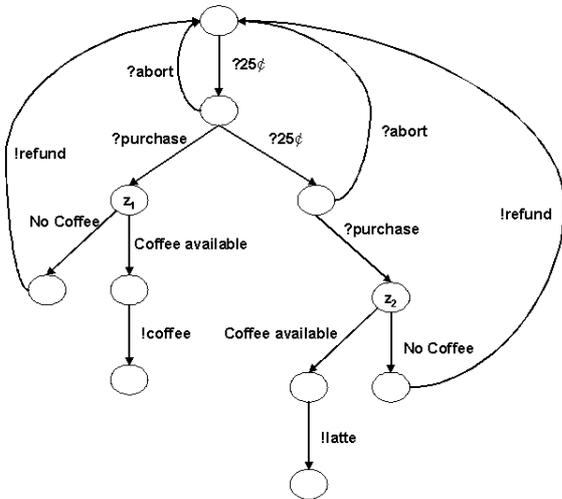
**Figure 5. Coffee Vending Machine**

The protocol, as defined in figure 5, assumes there is a very simple and affordable coffee vending machine that only accepts quarters, and it charges a quarter for a cup of regular coffee, and two quarters for a cup of latte. A customer can decide to cancel the order at any time before pressing the "purchase" button. If the customer cancels the purchase, the vending machine refunds and waits for the next order. The vending machine usually delivers coffee once a selection is made, but if there are no coffee beans left, it refunds and waits for the next customer.



**Figure 6. Language Equivalence**

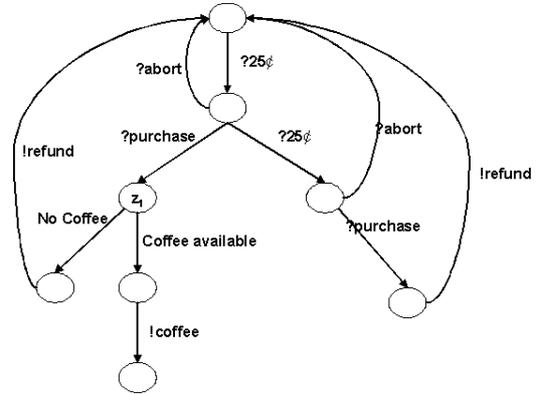
Following the same line of reasoning presented in this paper, we first examine language equivalence between the protocol and the implementation. The implementation of figure 6 is language equivalent to the participant *vending machine* in the protocol, since both accept the same language. However, if the machine moves to state  $Y_1$  after the customer inserts the first quarter, the customer is out of luck, since the only possible transition is to abort the transaction. Clearly, language equivalence is not strong enough for verification purpose.



**Figure 7. Violation of Strong Equivalence**

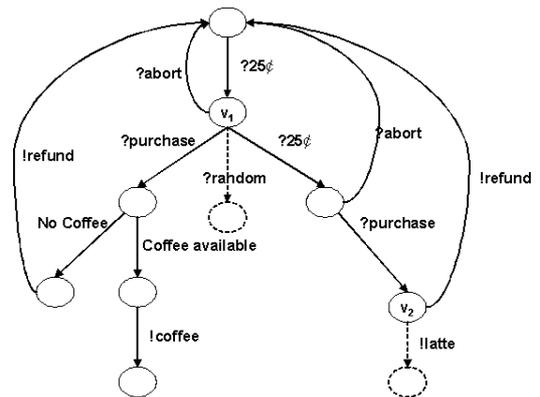
Not only do transitions, or the accepted language, need to match, but also the states these transitions lead to. This leads us to examine strong equivalence. Hence if our vending machine runs an internal routine to check whether there are coffee beans available in the grinder as state  $Z_1$  and  $Z_2$  in figure 7, we violate the strong equivalence relation, due

to the introduced states and internal transitions. However, the vending machine seems to operate fine as far as customers are concerned.



**Figure 8. Violation of Weak Equivalence**

In order to deal with only the observable interactions, that is the messages exchanged between participants, we consider weak equivalence. Weak equivalence ignores internal transitions and states, and focuses only on message exchange transitions. This relaxation takes care of the problem related to states  $Z_1$  and  $Z_2$  as discussed above, but what if the vending machine only sells regular coffee but not latte as shown in figure 8? Obviously, the customer would not be able to buy latte from this machine, since  $!latte$  transition is not implemented. This particular implementation would not be weakly equivalent. However, it does not violate the "Coffee Vending Machine" protocol in any way!



**Figure 9. Operational Compatibility**

Hence there is a need for further relaxation, and operational compatibility is introduced. The definition of operational compatibility is based on weak equivalence, com-

bined with an observation that at each state, the observable transitions from that state can only be either sending or receiving but not both. It is found that allowing more options for receiving states and less options for sending states will still keep an implementation operational compatible with the protocol. For a regular coffee vending machine as shown in figure 9, if state  $V_1$  is implemented to not only accepts another 25 cents, abort and purchase button, but also random button, since the front panel of the vending machine does not have random button, this addition would never cause any violation to the protocol. On the other hand, in state  $V_2$ , since this is a coffee vending machine that only offers regular coffee, customers would never get latte. But this implementation is perfectly fine as far as a participant of this protocol.

## 10. Conclusion and Future Work

Protocols define how participants should interact with their counterparts, and is of great value especially when conducting more complex transactions. In this paper, we investigate what the sufficient condition is when verifying whether the implementations of software agents adhere to the established protocols, and provide a verification algorithm with reasonable complexity. Using formal definitions, we started with the most intuitive yet overly restrictive notion of strong equivalence between the automata defined in the protocol and the actual implementation, and gradually relaxing unnecessary restrictions until coming to the definition of operational compatibility over weak equivalence.

This finding allows verification of software implementation against proper protocols to be automated. It allows web services to identify and interact with others that speak the same language. Using the same notion and algorithm, one of the ongoing projects of the Pervasive Computing Laboratory in the University of Florida is the safe guarding mechanism to prevent disruption of existing services in a smart space when introducing new software artifacts, by first verifying whether the new service bundles are well behaved according to the existing protocols before granting them entrance and access to the environment.

The mechanism presented here could be incorporated in UDDI [7]. We could extend the repository to include protocol specifications. Webservices could be given a certificate that proves that they are operationally compatible with a specified protocol. When a client wants to interact with a webservice the certificate is verified before the interaction commences. This assures that the webservice will not fail due to protocol failures.

We envision a new programming language based upon conversations as an extension to the object oriented programming paradigm. The language should allow us to specify interaction protocols and agents that participate in the

protocol. The operational compatibility allows us to verify whether the programmed agent is a proper participant of the protocol. Upon compilation we could create a certificate that proves the agent is a operational compatible with the specified protocol.

## References

- [1] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.
- [2] A. Helal, W. Mann, H. Elzabadani, J. King, Y. Kaddourah, and E. Jansen. Gator tech smart house: A programmable pervasive space. *IEEE Computer magazine*, pages 64–74, March 2005.
- [3] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, 2001.
- [4] F. Lin, P. Chu, and M. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *ACM*, 18:126–13, 1988.
- [5] B. Meyer. *Eiffel The Language*. Prentice Hall, 1992.
- [6] R. Milner. *Communicating and Mobile Systems: the  $\pi$ -Calculus*. Cambridge University Press, May 1999.
- [7] UDDI. *Universal Description, Discovery, and Integration of Business for the Web*, October 2001. URL: <http://www.uddi.org>.
- [8] H. van der Schoot and H. Ural. A uniform approach to tackle state explosion in verifying progress properties for networks of cfsms, 1996.
- [9] M. Yuang. Survey of protocol verification techniques based on finite state machine models. In *Proceedings of the Computer Networking Symposium*, pages 164–172, 1988.