TOWARDS FLEXIBLE CONVERSATION PROTOCOL GENERATION AND VERIFICATION: A SOCIAL COMMITMENT APPROACH

Martin T. Press. *PCSE, Christopher Newport University. 1 University Place, Newport News, Virginia USA 23606*
mpress2@cnu.edu

Roberto A. Flores. *PCSE, Christopher Newport University. 1 University Place, Newport News, Virginia USA 23606*
roberto.flores@cnu.edu (corresponding author)

ABSTRACT

We present a framework supporting the design of generic and flexible conversation protocols for multi-agent systems. This framework uses a social commitments approach in which descriptions are transformed into finite state machines for subsequent embedding in software agents. As part of this process, it provides a tool to verify the correctness of protocols in terms of their completeness and resolution of social commitments. The framework includes object-oriented concepts such as extension and aggregation to aid users reuse protocols during their design.

KEYWORDS

Agent communication, conversation protocols, social commitments.

1. INTRODUCTION

The heterogeneous nature of agents in open environments means that no assumptions can be made about their inner workings. This constraint is particularly acute in their communications—the use of messages and protocols—since these communications must be understood in terms that are external to agents.

At the higher level, a protocol can be seen as states and transition events, where events are message exchanges and states indicate goal advancement at a particular point between these exchanges. Currently, most protocols are developed statically at design time and cannot easily change at runtime. This lack of flexibility makes them impractical in open environments where agents must adapt their communications according to the state of the environment and to their particular abilities and preferences.

In this view, our goal is to provide a framework where flexible protocols are formalized and verified at design time. In general, there are two requirements when developing protocols: they should have goals and they should be verifiable. On the one hand, goals are ascribed to protocols at design time (Miller and McBurney, 2008). These goals can be modeled as the end states a designer intends a protocol to reach. On the other hand, verification happens when protocols are proven to reach a goal state on every run (Gouda, 1993). The verification of flexible protocols differs from that of static protocols in that the transitions and intermediate states toward goal states may vary during different runs of the same protocol. Accordingly, a key requirement to verify flexible protocols is ensuring that they can reach an end state regardless of variations between runs.

With the aim of flexible protocol verification in mind, we present a framework where flexible protocols can be defined in terms of their actions and goals, and whose correctness is verified by identifying that their end states are reached on every run.

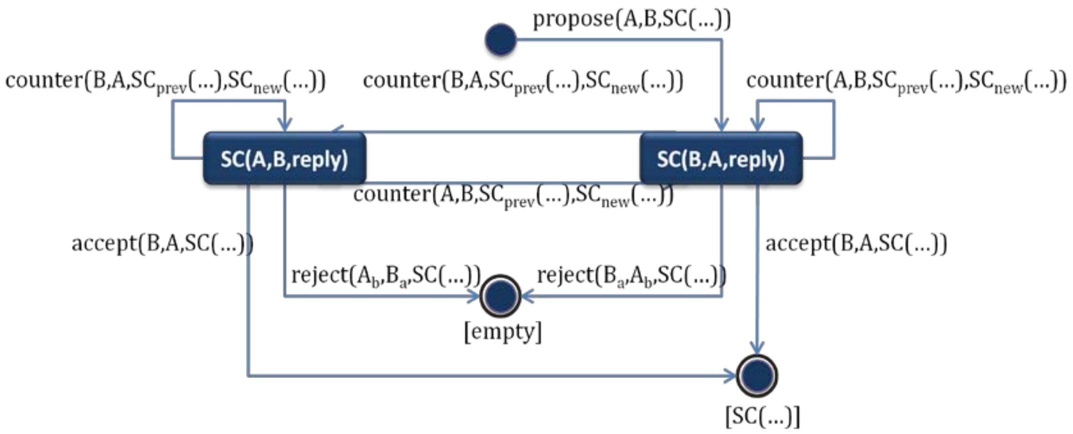
In a nutshell, the framework allows designing generic flexible protocols that can be transformed into working non-generic protocols for later implementation in agents. In addition, it supports protocol specialization as a way to ease the design process and encourage reusability. These topics are discussed in the remainder of the article as follows: Section 2 gives a brief overview of the framework's foundations and its extensions. Section 3 gives an overview of the protocol generation features of our framework. Section 4 overviews the design, generation and verification process with the aid of a use-case. Section 5 gives a view of related work in the field of protocol generation, and Section 6 presents conclusions and future work.

2. PROTOCOL FOR PROPOSALS

One way to define conversation protocols is using social commitments (Yolum, 2007, Flores & Kremer, 2005). In this view, commitments are added and discharged as messages are exchanged between agents.

A *social commitment* is an obligation put upon an agent to perform an action (Verdicchio & Colombetti, 2003). Each commitment involves two agent roles: *creditors* (who benefit from the commitment) and *debtors* (who are responsible to satisfy it). Commitments can be in one of several states, such as active, fulfilled and violated (Flores, Pasquier & Chaib-draa, 2007). A commitment becomes *active* once it is adopted by its debtors, who are responsible to satisfy its conditions, and becomes *fulfilled* and can be discharged if its creditors accept that its conditions have been satisfied; otherwise the commitment remains active until its conditions cannot be satisfied (e.g., its expected completion time has expired), making it become *violated.*

Our framework uses a meta-protocol for building conversation protocols using social commitments called the *protocol for proposals* (PFP) (Flores and Kremer, 2005). PFP is a mechanism that allows agents to the agreed uptake and discharge of commitments (thus supporting agent autonomy) and their negotiation as conversations progress (thus supporting flexible conversations at runtime). As shown in Figure 1, the initial message in a PFP instance is *propose*. Proposals trigger the only conversational norm in the PFP: that the receiver of a propose message is committed to reply. Replies have the form of either an *accept*, a *reject* or a *counter-propose* message. By defining a counter-propose as a rejection of its immediate propose and a propose with an alternate commitment, the PFP guarantees the continuity of the conversation until an accept or a reject message is issued. Accepting a proposal cements the (adoption or discharge of the) proposed commitment. Rejecting a proposal leaves agents in the same state they were prior to the PFP conversation.



propose(A,B,SC(...))
counter(B,A,SC_{prev}(...),SC_{new}(...))
counter(B,A,SC_{prev}(...),SC_{new}(...))
counter(A,B,SC_{prev}(...),SC_{new}(...))
SC(A,B,reply)
SC(B,A,reply)
counter(A,B,SC_{prev}(...),SC_{new}(...))
accept(B,A,SC(...))
accept(B,A,SC(...))
reject(A_b,B_a,SC(...))
reject(B_a,A_b,SC(...))
[empty]
[SC(...)]

Figure 1. State diagram of the Protocol for Proposals

The PFP was originally defined to operate on a single commitment, thus limiting the type of protocols it can create. Our work extends it by allowing sequential and alternative actions to occur through *composite social commitments.* As shown in the simplified BNF notation below, composites use *or* and *and* logical operators to link commitments.

<*PFPCommitment*> $\rightarrow$ (**add** | **remove**) <*CompositeCommitment*>
<*CompositeCommitment*> $\rightarrow$ <*SocialCommitment*> { <*Logical*> <*CommitmentTerm*> }
<*CommitmentTerm*> $\rightarrow$ (<Composite*Commitment*>) | <*CompositeCommitment*>
<*SocialCommitment*> $\rightarrow$ **SocialCommitment(** { *creditor* }$^{+}$**,** { *debtor* }$^{+}$ **,** { *action* }$^{+}$ **)**
<*Logical*> $\rightarrow$ (**and**$_{set}$ **|** **and**$_{sequence}$ **|** **or**$_{apriori}$ **|** **or**$_{aposteriori}$)

Commitments in a composite can be linked by *and* (conjunction) and *or* (disjunction) logical connectors. Disjunction is either *a priori*, if the debtor is required to indicate at acceptance time which choice of commitments it will pursue; or *a posteriori*, if the debtor has the option of defining its choice of commitments either now (at acceptance time) or later (at the time of fulfillment). Conjunction can be either a *set*, if commitments are fulfilled in any order; or a *sequence*, if they must be fulfilled in the indicated order.

For example, a customer and a vendor can use a composite commitment to buy an item, which can be defined as paying and receiving the item. If the vendor accepts these commitments and they were linked as a *set* then the vendor is allowed to choose whether to receive the payment first and send the item later, or deliver first and be paid later. Differently, if the commitments were linked as a *sequence* then the vendor could only proceed in the order indicated.

3. CONVERSATION PROTOCOLS FRAMEWORK

At a high level, our framework is composed of three processes: *action design*, *protocol generation & verification* and *protocol implementation*. Although in this article we cover notions of the first process (action design) we are mostly concerned with the second process (generation and verification). The third process (implementation) is left as future work. In brief, the generation & verification process receives objects and actions in a XML-based language we named *GenAct*, and allows designers to visualize, edit information and verify the correctness of protocols before saving them for later implementation.

In the following sections, we provide a few basic protocol definitions followed by *GenAct* examples.

3.1 Protocol Paths

Protocol instances can take one of several available paths at runtime. A *path* is a sequence of states and transitions in a protocol. We identify three different types of paths: *optimal*, *successful* and *exception*. An *optimal path* is the shortest path leading to the intended goal of the protocol, and where the *intended goal* is the end state favored by the protocol's designers. In PFP, an optimal path would have propose and accept messages only. Optimal paths do not have any reject (which make a protocol return to a previously agreed state) or counter-propose messages (which would increase the message count to reach a goal). A *successful path* is a path in which the intended outcome is achieved regardless of the messages required. Successful paths could include propose, accept, reject and counter-propose messages for as long as the goals of the protocol are reached.[1] An optimal path is a special case of a successful path with no counter-proposals or rejections. An *exception path* is any path not reaching the goal given rejections or unfulfilled commitments.

3.2 Protocol Types

We identify two types of protocols in the framework: *concrete* and *abstract*.

A *concrete protocol* is a protocol where there is a complete temporal ordering of states and transitions, and where all agent roles and actions are defined. The only aspect that could be left unspecified is the time

[1] Although it is easier to envision a successful path with counter-proposals (e.g., "Let's go to the movies at 4 pm" followed by "What about 7 pm instead?" and an agreement leads to the intended goal "Go to the movies together"), they can also include rejections. For example, if A and B have agreed that B will build a chair for A, and B attempts to discharge this commitment (e.g., "Here is the chair you requested") but A rejects ("But...it's missing a leg!") makes the state of the conversation fall to a previous stable state (i.e., B will build a chair for A). In this example, the conversational goal "B built a chair for A" can be reached if A agrees with a subsequent discharge attempt from B.

expected between messages, e.g., replying to a propose is expected within 5 seconds. In addition, time can be absolute or specified as an offset of the starting time of the protocol.

An *abstract protocol* is a protocol whose states and transitions are defined at design time but whose actions and sequencing are not. For example, a *Buy* abstract protocol may not specify what item will be bought, the amount that will be spent on it, whether the customer should pay before or after delivery of the item, or whether the item is paid with cash or a credit card.

Concrete protocols are created from abstract protocols whose values and allowed paths are all identified. We derive that a concrete protocol is a subset of its abstract protocol, and that an abstract protocol embodies all the features of its concrete protocols. In addition, our language promotes protocol and action reuse through extension (inheritance) and aggregation (adoption).

3.3 Designing Actions

We define an *activity* as a non-empty set of actions, which together define the goal of a protocol. Activities use object-oriented programming paradigm techniques such as inheritance and polymorphism.

3.3.1 Basic Hierarchy

Figure 2 shows the basic primitives in our language: *object*, *agent*, *social commitment* and *action*.

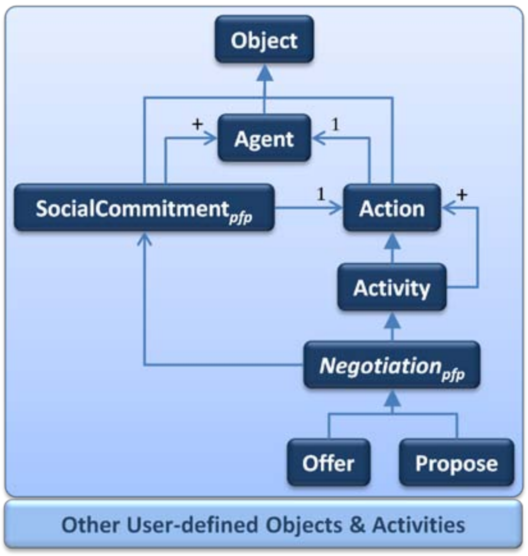


Figure 2. UML diagram of *GenAct* primitive types

An *object* is the basis for any entity, such as an *agent*, which is an actor in the system.

```
<Agent  type="Object" />
```

As shown in the below, *action* is an agent performance over a subject, and the basis for *activity* structures. Note that the tag <declare> is used for declaring new entities within a definition. Other available tags are <assign>, for binding entities in aggregated definitions, and the activity-only tags <activities>, to aggregate activities within an activity, and <protocol>, to indicate ordering of aggregated/inherited activities (examples to follow).

```
<Action type="Object">
  <declare>
    <Agent name="performer" />
    <Object name="subject" />
  </declare>
</Action>

<Activity type="Action" />
```

A *social commitment* is a directed obligation to perform an action. As such, they declare a creditor agent, a debtor agent and an action.

```
<SocialCommitment type="Object">
  <declare>
    <Agent name = "creditor" />
    <Agent name = "debtor" />
    <Action name = "action" />
  </declare>
</SocialCommitment>
```

Activities are negotiated through PFP action primitives such as *offer* and *propose*, which have a sender, a receiver and a commitment. An *offer* is an occurrence where the speaker is the debtor of the commitment, and a *proposal* is an occurrence where the hearer is the debtor of the commitment.

```
<Negotiation type="Activity" >
  <declare>
    <Agent name="sender" />
    <Agent name="receiver" />
    <SocialCommitment  name="commitment" />
  </declare >
</Negotiation>

<Propose type="Negotiation" >
  <assign>
     <commitment:creditor  name="sender" />
     <commitment:debtor  name="receiver" />
     <commitment:action:performer  name="receiver" />
  </assign>
</Propose>

<Offer type="Negotiation" >
  <assign>
    <commitment:creditor  name="receiver" />
    <commitment:debtor  name="sender" />
    <commitment:action:performer  name="sender" />
  </assign>
</Offer>
```

Using these types, designers can add their own activities and objects, such as currency and credit card.

```
<Currency  type="Object" />
<CreditCard  type="Currency" />
```

3.3.2 Defining Activities

Activities have four sections: *object definition*, *object assignment*, *sub-activities* and *protocol*.

In the *object definition* section (using the <declare> tag), objects are defined with a type and name and can be referenced in this and other activities. In the *object assignment* section (using the <assign> tag), objects are linked to existing objects across inherited activities and sub-activities. In the *sub-activities* section (using the <activities> tag), existing activities can be added to the current activity, and their ordering is defined in the *protocol* section (using the <protocol> tag). The *Buy* activity below exemplifies the use of these tags.

```
<Buy type="Activity" >
  <declare>
    <Agent  name="seller" />
    <Agent  name="buyer" />
    <Object  name="goods" />
    <Currency  name="currency" />
  </declare>
  <activities>
```

```
        <Offer name="pay" >
          <assign>
            <sender  name="buyer" />
            <receiver  name="seller" />
            <commitment:action:subject  name="currency" />
          </assign>
        </Offer>
        <Offer name="deliver">
          <assign>
            <sender  name="seller" />
            <receiver  name="buyer" />
            <commitment:action:subject  name="goods"/>
          </assign>
        </Offer>
      </activities>
      <protocol>
        <set>
          <pay />
          <deliver />
        </set>
      </protocol>
    </Buy>
```

As shown above, the <declare> section indicates that *Buy* involves a seller and a buyer agents, a generic type of goods and a currency object. The <activities> section adds the sub-activities *pay* and *deliver* (of type *offer*). This is illustrated in Figure 3, which shows both the aggregation of these activities and the inter-relationship between the objects they define. In particular, the sub-activity *pay* has an <assign> tag indicating that the buyer is the sender, the seller is the receiver, and the currency is the subject of the offer. By recalling the definition of offer, we also know that the seller is the creditor of the commitment, and the buyer is the debtor of the commitment and the performer of the action of sending the currency. Likewise, the sub-activity *deliver* has an <assign> tag indicating that the seller is the sender and the buyer the receiver of the goods; and from the definition of offer we also know that the seller is committed to the buyer to deliver these goods.

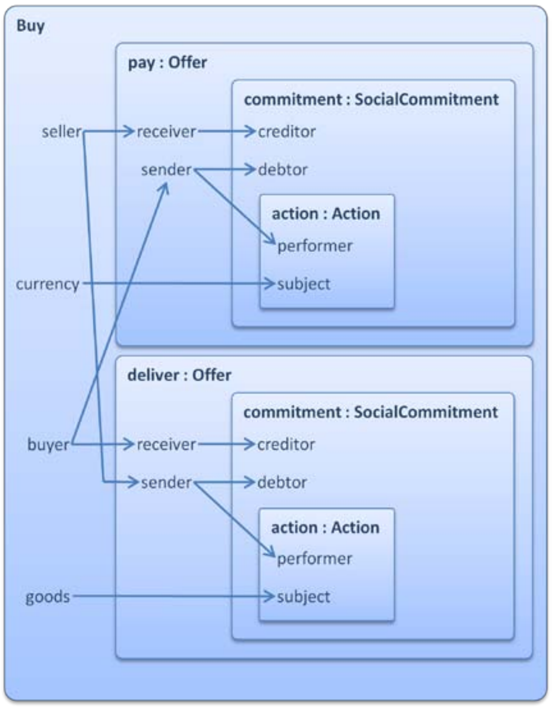


Figure 3. Sub-activities and the inter-relationship of their objects in the *Buy* activity

The <protocol> section specifies the ordering of sub-activities, in this case using the <set> tag to indicate that there is no ordering—either delivering could predate paying or vice versa, or they could occur concurrently; it nevertheless mandates that these sub-activities will happen as part of the activity. If ordering was needed, designers can create a new activity inheriting from *Buy*, for example, *PayAndDeliver* (shown below) in which the protocol section is overridden to mandate that inherited sub-activities unfold in a certain way, in this case that *pay* must go before *deliver*.

```
<PayAndDeliver type="Buy">
  <protocol>
    <sequence>
      <pay />
      <deliver />
    </sequence>
  </protocol>
</PayAndDeliver>
```

In brief, the protocol section currently supports the tags <choice>, <set> and <sequence>. The *choice* tag defines a divergent path of activities, where only one path can followed; the *sequence* tag lists activities in the order in which they must be executed; and, the *set* tag indicates inclusion of activities but not their order of execution (thus any sequence of occurrence is permitted).

As exemplified above, *GenAct* promotes reuse through extension (inheritance) and adoption (aggregation) mechanisms, where all defined objects are visible within the activities in which they are reused. In addition, inherited objects can be overridden with other (type compatible) objects by using the same name as the one in the parent activity. For example, *BuyWithCreditCard* (shown below) can be defined as an activity that inherits from *Buy* and overrides *currency* with an object of type *CreditCard* (which was defined earlier as a subtype of *Currency*).

```
<BuyWithCreditCard type="Buy" >
  <declare>
    <CreditCard name="creditCard"/>
  </declare>
  <assign>
     <currency name="creditCard" />
  </assigns>
</BuyWithCreditCard>
```

3.4 From Actions to Protocols

By defining PFP negotiations in terms of social commitments and activities, we can map actions into finite state machines of PFP instances. In particular, any action is translated into a pair of PFP instances, one in which the commitment to an activity is added, and another one in which the commitment is deleted. Moreover, adoption (or deletion) of a composite commitment implies adoption (or deletion) of all its enclosed commitments; and composites can be removed if the negotiation of their commitments fails.

3.4.1 Finite State Machines

Control tags in activities support the generation of flexible PFP finite state machines.

The *set* tag indicates a divergent path of PFP instances, resulting in paths with all permutations of actions in the tag. Each path can have two outcomes: one in which all commitments are satisfied and the end state is reached; another in which there is at least one case where a commitment is not satisfied. In the former, the composite commitment is removed after all its commitments are naturally fulfilled. In the latter, an exception path is taken to remove the composite, and thus all its commitments. In the case of the *Buy* activity, we defined it with a *set* tag with activities *deliver* and *pay*. As shown in Figure 4, the initial PFP would add the composite *Buy* and its commitments in *pay* and *deliver*. After this PFP there are two paths where the protocol could go: one where paying comes first and delivering comes later; and the other where delivering comes first and paying later. Exception paths are shown as dashed lines between PFP instances to remove the composite.

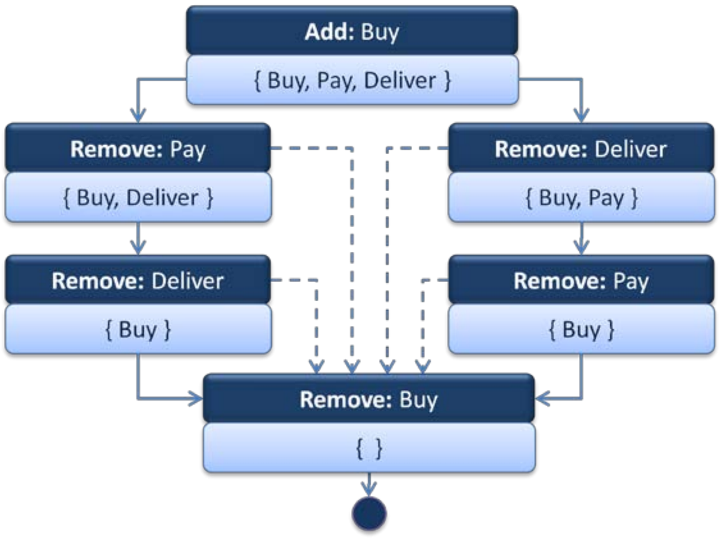


Figure 4. Example of a PFP protocol using the *set* control tag

The *choice* tag has mutually exclusive paths, each with exception paths to remove the composite in case of failure. Figure 5 shows a *BuyTrainOrAirTicket* activity with the choice of buying either a train or an airplane ticket. When using an *a posteriori* connector, both commitments are added but only one is to be fulfilled. In this example, if the PFP for buying a bus ticket is successful then the commitment for buying an airplane ticket is also discharged (without fulfillment). The exception path behaves in the same way as in the earlier *set* example, removing the composite and all its enclosed commitments (if any remain).

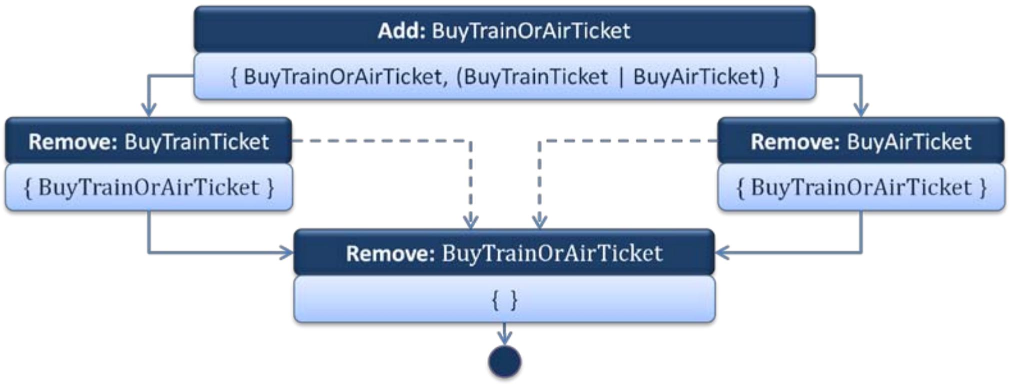


Figure 5. Example of a PFP protocol using the *choice* control tag

Lastly, the *sequence* tag states the linear order of the actions it contains. One example activity is that of registering for classes, in which students must first offer their resolved schedule of classes to an advisor before this advisor hands to the student the PIN required for registration. This activity is defined as follows:

```
<Register type="Activity" >
  <declare>
    <Agent name="student" />
    <Agent name="advisor" />
    <Object name="schedule" />
    <Object name="PIN" />
  </declare>
  <activities>
    <Offer name="offerSchedule ">
      <assign>
        <sender name="student" />
        <receiver name="advisor" />
```

```
            <initial:action:subject name="schedule" />
          </assign>
        </Offer>
        <Propose name="requestPIN">
          <assign>
            <sender name="advisor" />
            <receiver name="student" />
            <initial:action:subject name="PIN"/>
          </assign>
        </Propose>
      </activities>
      <protocol>
        <sequence>
          <offerSchedule />
          <requestPIN />
        </sequence>
      </protocol>
    </Register>
```

Figure 6 shows this activity and its corresponding exception paths.

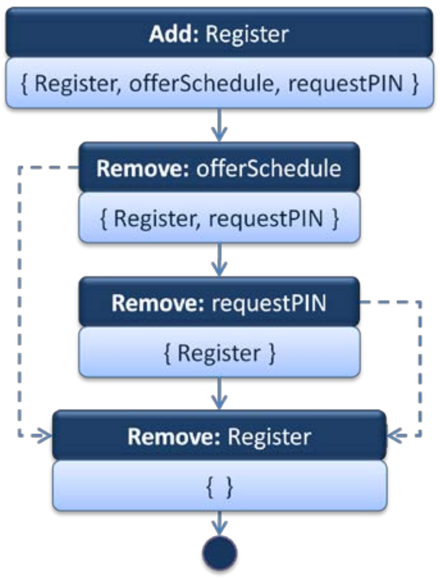


Figure 6. Example of a PFP protocol using the *sequence* control tag

Lastly,we show the language's ability to extend protocols by creating a *RegisterHonors* activity that extends from *Register*. In this (hypothetical) activity, students have the option of either getting their PIN from their advisor (as any student would do) or they can get an approval signature from the Dean. Such an activity is defined below (note the use of the tag <super /> to include the protocol inherited from *Register* as part of the current protocol).

```
<RegisterHonors type="Register" >
  <declare>
    <Agent name="Dean" />
    <Object name="signature" />
  </declare>
  <activities>
    <Propose name="requestSignature">
      <assign>
        <sender name="student"  />
        <receiver name="dean" />
```

```
            <initial:action:subject name="signature"/>
         </assign>
      </Propose>
   </activities>
   <protocol>
      <choice>
         <requestSignature />
         <super />
   </choice>
   </protocol>
</RegisterHonors>
```

3.5 Conversation Protocol Verification

The protocol verification part of the process has two purposes: to check that protocols are semantically valid, and to ensure that commitments are resolved at their end states.

In agent based systems, there are three requirements for a protocol to be considered semantically valid (Yolum, 2007): (a) that the protocol is free of cyclic runs, (b) that it is deadlock free, and (c) that it has valid end states. A protocol is non-cyclic if there is at least one transition exiting a cycle of states. This prevents a protocol from containing a loop of transitions that can never reach an end state. A protocol is deadlock free if agents are not allowed to create commitments that contradict each other. For example, an agent may agree to wash a car but it has previously made a commitment not to wash it, which would make it impossible for the agent to fulfill both in overlapping contexts. Lastly, a protocol has end states if it has states with no outgoing transitions.

3.5.1 Non-Cyclic and Complete Protocols

Using of PFP as the building block of protocols alleviates verifying that they are non-cyclical. At every node of a PFP there are two internal states, one with a commitment to reply; and in each there is at least one transition to an exit state (beside an acceptance from the receiver, a propose can be counter-proposed or rejected by both the hearer and the speaker, if it changes its mind). Even with counter-proposals adding cycles, all of them are escapable by any agent at any time through *reject* or *accept* messages. Therefore, since PFP instances can always reach an exit state, we can be sure that we can use them without checking for cycles within generated protocols.

On the other hand, to check that protocols based on the PFP are valid we perform a depth-first search with loop detection, verifying whether the node reaches an end state. Otherwise the protocol is cyclic and invalid.

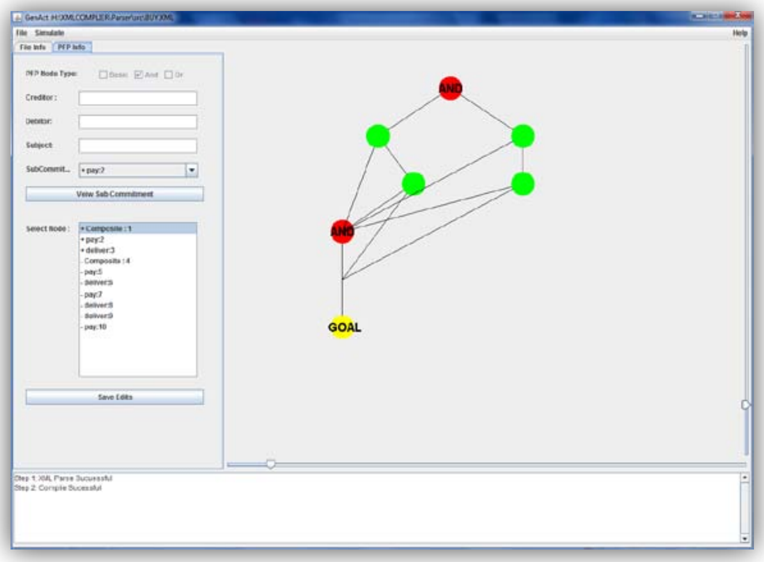
3.5.2 Resolved Commitments

To verify that a protocol is correct from the standpoint of commitments can be done by simulating runs of the protocol through all paths (excluding counter-proposals) to show that all commitments are satisfied when the end state is reached. By using a commitment store, a simulator adds and removes commitments as it advances through a path. To succeed, the store should be empty when the simulator reaches the end goal for all paths, and no cases should exist in which a non-existent commitment is attempted for removal.

4. SYSTEM IMPLEMENTATION

4.1 Design and Generation

As it was shown earlier, a *Buy* activity is defined in XML as an activity that uses *pay* and *deliver* sub-activities within a *set* tag. Once defined, this activity can be loaded into the *GenAct* program (shown in Figure 7), where the activity is parsed for syntactical errors and rendered as a graph.



PFP Node Type: [ ] Basic [✓ And] [ ] Or
Creditor :
Debtor:
Subject:
SubCommit... • pay:2

Select Node :
• Composite : 1
• pay:2
• deliver:3
- Composite : 4
- pay:5
- deliver:6
- pay:7
- deliver:8
- deliver:9
- pay:10

```
Step 1: XML Parse Sucuessful
Step 2: Compile Sucessful
```

Figure 7. The *GenAct* GUI software interface

Once the protocol has been syntax-checked, it will be in an abstract state until all involved creditor and debtor agents are specified. As shown in the figure, the program displays concrete nodes (in green) and abstract nodes (in red). Clicking on any of these nodes allows correcting or adding information. The verification process can only begin when all nodes have been changed from abstract to concrete.

4.2 Verification

As mentioned earlier, a depth-first search is used to check that all PFP nodes can reach an end state. This search is not enough to verify that the protocol is correct, since abstract nodes could be incorrectly edited and may not necessarily fulfill all commitments. This may lead to two possible execution errors: agents trying to remove a commitment they do not have, or agents whose state has a commitment leftover. In these cases, the simulator can check that all paths end with an empty commitment states for all interacting agents, reporting any non-relieved commitments. With this feedback, designers can reedit nodes to ensure all commitments are satisfied, before saving the protocol as XML data that agents can be programmed to follow.

5. RELATED WORK

In this section we describe and compare against similar frameworks dealing with protocol specification and verification.

5.1 Meta-Protocols

McGinnis (2006) introduced a BDI game theory framework to generate agent-based protocols, in which a set of information-seeking games controls the structure of protocols. In this framework, agents participate in a protocol only if they have agreed to enter it. Our framework takes a similar approach but simplifies control. Instead of high-level protocols for path control, PFP imbeds control into negotiated commitments, which agents can reject if they do not find them favorable. In addition, we take a social commitment (rather than a BDI) approach, which can support protocol correctness without having to inspect the internal states of agents.

Another meta-protocol approach is the voting system described by Artikis (2009). In this system, each protocol has a number of variables (called degrees of freedom) that can be voted by agents at runtime. These variables are defined at design time and cannot be changed during execution. Also a degree of institutional authority is given to the agents at different levels. Agents at higher levels can participate in higher levels of nested voting, which leads to their imposition of protocols to lower-level agents. Differently, PFP agents are not subject to institutional constraints and thus are more portable and flexible to implement.

5.2 Social Commitments

Yolum and Singh (2002) describe a framework for social commitment protocols, called commitment machines, in which protocols do not have a definitive start state; instead, any state can be a starting state as long as the commitment states of agents match the states of a protocol. The framework does not specify all the possible transitions or runs of protocols; instead it defines the meaning of each state (in terms of commitments), the actions allowed, and the new states after these actions. Our framework is more robust since protocol states naturally emerge from the negotiation of commitments, and are not mandated by design. Mallya and Singh (2005) propose a mechanism, called splicing, to attach protocols to other protocols at runtime, allowing them to return to their normal execution in case of failure. Splicing has the disadvantage of needing a predefined (at design time) library for exceptions. We have similar mechanisms on exception paths and polymorphism, since the former allow agents to recover from failure, and the latter allows child activities to override the behavior defined in their parent without affecting the rest of the protocol. In addition, we are considering a *failure* tag in our language (future work) that would provide recovery paths in cases of failure. Lastly, we recently found work by Kremer (2012) in which protocols are defined as extensible LISP modules in the Collaborative Agents Systems Architecture (CASA). Protocols are based on conversation templates that indicate the policies and standard behavior that apply to message types. Similarly to our approach, protocols are built by reusing other protocols, and the states of the sub-protocols are synchronized according to the messages received and the state of the enclosing protocol. This approach has the advantages of allowing conversation policies and LISP executable code as part of the protocols. One possible advantage of our approach is that it caters to heterogeneous systems where executable languages are not mandated.

6. CONCLUSION AND FUTURE WORK

Our framework has room for practical and theoretical improvements. On the theoretical side, we need to support language extensions at design time and dynamic protocols at runtime. On the practical side, improvements are still needed on the framework's workflow and the program's user interface.

On the theoretical side, we have noticed that some protocols should leave commitments after their execution, and some others require agents to have commitments before these protocols take place. For example, agents using a getting-married protocol end up with marriage commitments when the protocol ends. Conversely, a getting-divorced protocol will require that agents have the commitments gained through marrying each other prior to its execution. Commitments intended to remain at the end of a protocol are part of the post-conditions of protocols, and can be identified as the commitments adopted and not discharged at the end of successful paths. In addition to commitments, pre- & post-conditions may include objects; for example, a customs agent admitting a traveler into the country would need a passport and visa as pre-condition objects, and would result on an official stamp as its post- condition. Adding pre- & post-conditions would require adding XML tags to our language and extending the verification process (which currently validates that a protocol discharges all commitments) to check that successful paths consistently result on the intended protocol post-conditions. Other language extensions are the *optional* and *failure* tags. The *optional* tag would indicate secondary but complementing activities in a protocol; for example, a movie-night activity may indicate buying popcorn and soda as optional activities. This tag would substantially reduce the number of paths that would be generated otherwise if using *or* (alternative) tags. The *failure* tag would indicate paths that could be followed to recover from a failure by defining actions that can only be pursued if the intended path cannot be completed; for example, in a class-override activity, a student could request permission to the Dean to get into a class only if the instructor is not available. A third extension is allowing preferences in alternative paths and in *set* activities. By indicating preferences (either at design or runtime) agents could have a mechanism to influence (but not mandate) the flow of a protocol and its activities; for example, a protocol to buy a train or an airplane ticket would indicate that the proposing agent would prefer that a train ticket be bought although the protocol's goal would also be satisfied if the airplane ticket was bought instead.

One other research avenue we will pursue has to do with the compatibility of protocols in inherited activities. At the present time, declaring a protocol section in an activity effectively overrides the behavior in the parent activity leaving no assurances that the new protocol complies with the intention in the older. We plan to explore pertinent disciplines (e.g., graph theory) to identify possible models or methodologies that could improve our approach.

On the practical side, there is room to improve the GUI program in several fronts, such as supporting action design (the editing of XML tags is done by manually writing to text files), improved usability (nodes are displayed and their data can be edited but there is no graphical support to link objects and actions in activities or to edit nested activities) and a thorough support to our language (currently only the alternative a priori *or* and the sequence *and* tags are supported).

A further area of exploration is the deployment of protocols in agents, either by designing an environment in which agents can be programmed to follow a protocol or can generate their own protocols dynamically; in this latter case, when a new protocol is required, agents could dynamically generate one and propose it to other potential interacting agents using existing framework mechanisms.

To conclude, in this article we presented an action-based framework for designing multi-agent protocols. This framework is based on the PFP meta-protocol and a XML language called *GenAct* in which designers can generate correct protocols that are verified in terms of completeness and resolved social commitments.

ACKNOWLEDGEMENTS

We are thankful to the National Science Foundation grant No. 0841295 for partially funding our effort, to the reviewers and organizers of ISA 2011 for their support, and especially grateful to Robert Kremer at the University of Calgary for sharing his thoughts, invaluable advice and encouragement to improve this work.

REFERENCES

Artikis, A., 2009, Dynamic Protocols for Open Agent Systems. *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, Budapest, Hungary, ACM, pp. 97-104.

Flores, R.A., and Kremer, R.C. 2005. Commitment-based Conversation Protocols. *Proceedings of the 4th IASTED International Conference on Computational Intelligence (CI 2005)*, Calgary, Canada, pp. 147-152.

Flores, R.A., Pasquier, P. and Chaib-draa, B., 2007. Conversational Semantics Sustained by Commitments. *Autonomous Agents and Multi-Agent Systems*, Volume 14, Number 2, Springer-Verlag, pp. 165-186.

Gouda, M.G., 1993. Protocol Verification Made Simple: A Tutorial. *Computer Networks and ISDN Systems*, volume 25, number 9, Elsevier Science Publishers, pp. 969-980.

Kremer, R.C. (2012) “Defining Conversation Protocols in CASA”. To be published.

Mallya, A.U. and Singh, M.P., 2005. Modeling Exceptions Via Commitment Protocols. *Proceedings of the 4th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, Utrech, The Netherlands, ACM, pp. 122-129.

McGinnis, J.P., 2006. *On the Mutability of Protocols*. Ph.D. thesis. University of Edingburg. Last accessed: December 2010. http://www.cisa.inf.ed.ac.uk/ssp/pubs/mcginnis_phd.pdf

Miller, T. and McBurney, P., 2008. On Illegal Composition of First-Class Agent Interaction Protocols. *Proceedings of the 31st Australasian Conference on Computer Science (ACSC '08)*, Wollongong, Australia, Australian Computer Society, pp. 127-136.

Verdicchio, M. and Colombetti, M., 2003. A Logical Model of Social Commitments for Agent Communication. *Proceedings of the 2nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003)*, Melbourne, Australia, ACM, pp. 528-535.

Yolum, P., 2007 Design Time Analysis of Multiagent Protocols. *In Data and Knowledge Engineering*, volume 63, number 1, Elsevier Science Publishers, pp. 137-154.

Yolum, P. and Singh, M.P., 2002. Commitment Machines. *Proceedings of the 8th International Workshop on Intelligent Agents VIII (ATAL '01)*, Springer-Verlag, pp. 235-247.