Evolving Protocols and Agents in Multiagent Systems

Scott N. Gerard
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
sgerard@us.ibm.com

Munindar P. Singh
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
singh@ncsu.edu

ABSTRACT

We consider multiagent systems that involve two or more business partners interacting via autonomous software *agents*. A *(business) protocol* describes the messages exchanged by the agents in high-level terms. Such systems pose a major challenge with requirements evolution. Current approaches couple agent and protocol designs, requiring coordinated changes. In contrast, we propose an approach that decouples agent and protocol designs, while maintaining interoperability. We build on the well-known architectural construct of an *interceptor*. We introduce *interaction refactorings* to transform interactions in response to evolving requirements, with each refactoring incrementally changing agents, interceptors, and the protocol. We identify three main forms of requirements evolution and propose an extensible library of refactorings that help address each form. We demonstrate the approach through examples and a JADE prototype.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification

General Terms

Design

Keywords

Refactor; Interaction; Interceptor; Communication protocols; Agent communication; Commitments

1. INTRODUCTION

We consider cross-organizational multiagent systems that arise when two or more business partners interact, for example, to carry out complex service engagements. Each business partner implements a software *agent* that appears to the rest of the system to be autonomous and active (both proactive and reactive). To facilitate the interoperation of the partners' agents, such systems are often built using *(business) protocols* that specify the messages that the agents may exchange along with any constraints on such messages.

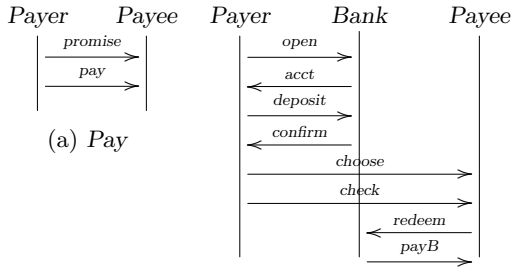
Appears in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013), Ito, Jonker, Gini, and Shehory (eds.), May, 6–10, 2013, Saint Paul, Minnesota, USA.*

Although such protocols are valuable for engineering, they result in architectural coupling: Designers cannot deploy a new protocol until all parties agree and their agents are modified accordingly. In general, protocol and agent interfaces must change together.

In essence, the decentralized nature of multiagent systems makes it difficult to handle evolving requirements since any change appears to demand bulk (concurrent and coordinated) updates, which are precisely ill-suited for a decentralized system. In today's practice, the business partners negotiate such updates by personal communication. The traditional approach faces a vicious cycle. First, without numerous agent implementations that exploit a new protocol, protocol adoption is hindered. Second, without wide protocol adoption, agent designers are little motivated to implement a new protocol.

1.1 Problem: Requirements Evolution

Agents are designed by agent designers and protocols are designed by protocol designers. We assume agent and protocol designers are distinct. When requirements change, to break the vicious cycle of the traditional approach, we desire a system where (1) concurrent and coordinated deployments are unnecessary; (2) agents can interoperate using an evolved protocol, without agent code changes; (3) each designer can work independently; and (4) designers can collaborate when necessary.



(a) *Pay*

(b) *PayViaCheck*

Figure 1: Example protocols: possible enactments.

To illustrate our proposal, we introduce two running payment protocols, *Pay* and *PayViaCheck*, with Figure 1 showing suggestive enactments. In *Pay*, *Payer* can promise to pay *Payee*, creating a commitment. Once committed, *Payer* pays at some future point. *PayViaCheck* is similar, except *Payer* must first open an account with *Bank*. At any time,

Payer can make confirmed deposits and send *Payee* a check that it may redeem for payment at *Bank*. We identify three forms of requirements evolution in the above setting.

***Protocol Designer Independence*(PDI)** Assume the agents initially employ *Pay*. A new legal requirement arises to ensure all payments are traceable, which *PayViaCheck* addresses. How can a protocol designer evolve protocol *Pay* to *PayViaCheck* to address this requirement without having to ask that all agents be concurrently updated?

***Agent Designer Independence*(ADI)** Assume the agents initially employ a payment protocol that supports travelers checks and other forms of payment. A new requirement arises for a specific *Payer* agent to reduce costs by ceasing to use travelers checks. How can this agent simplification result in protocol simplification?

***Designer Collaboration*(DC)** At times, designers must collaborate, with one agent's changes propagating to other agents. DC changes are an integral element of the protocol simplification just mentioned.

1.2 Approach: Refactoring Interactions

Our approach builds on the time-honored architectural abstraction of an *interceptor* or Chain of Responsibility pattern [7, 15]. Extending this, we show how to construct interceptors modularly in a rule-based manner from logical specifications of refactorings. Specifically, each interceptor is expressed as a series of reaction rules that are triggered by a message and which may refer to the interceptor's internal state. An *interceptor chain* is an ordered list of zero or more interceptors, that mediates all message flow to and from its agent. Incoming messages pass through an interceptor chain before arriving at the business logic component of an agent and outgoing messages likewise pass through the same interceptor chain in reverse order.

Given one or more agents that use a protocol, designers can incrementally *refactor* the agent and protocol interactions while preserving interoperability of the agents. A refactoring defines a set of coordinated, incremental changes to agents, interceptor chains, and the protocol. We provide an extensible library of refactorings from which designers may select and apply one or more refactorings to implement requirement changes. We partition refactorings into three groups based on the requirements evolution problem they address:

PDI For example, to evolve *Pay* to *PayViaCheck*, the protocol designer adds redemption processing by adding interceptors to convert each *pay* message to the message sequence *deposit*, *confirm*, *check*, *redeem*, and *payB*.

ADI Our approach enables agent and agent interface changes: (1) moving (internalizing or externalizing) functionality between the agent implementation and the interceptor chain, and (2) an agent declaring it will not send messages in cases enabled by the protocol.

DC These refactorings enable reorganizing, optimizing, and simplifying an interceptor chain.

Contribution and Organization

Our main contribution is the concept of *interaction refactorings* that enable independent and incremental evolution of interactions, decoupling the efforts of agent and protocol designers. Section 2 describes our enabling framework. Section 3 introduces representative refactorings and the underlying interceptor architecture. We apply refactorings to our example protocols in Section 4. We evaluate a prototype implementation in Section 5. Section 6 concludes with comparisons and a discussion of related work. This paper is previously unpublished except Section 2.1, which provides essential background.

2. APPROACH ILLUSTRATED

For brevity and clarity, we introduce the key concepts and syntax for our approach via examples.

2.1 Protocol and Commitment Background

A commitment $\mathsf{C}_{\{debtors\},\{creditors\}}(ant, csq)$ means the set of debtor roles commits to the set of creditor roles to make the consequent (*csq*) true whenever the antecedent (*ant*) is true, following Gerard and Singh [8]. And, a protocol is a set of guarded statements:

sender → *receiver*: [*guard*] *message* means{*meanings*}.

The sender *must not* send a *message* when the Boolean, public *guard* expression is false; it *may* send it when the *guard* is true.

Each message's meaning is a set of actions on propositions (SET and CLR) and commitments (CREATE, TRANSFER, RELEASE, and CANCEL). For example, in *Pay*, we have

Payer → *Payee*: [*promise*] *payMsg* means {SET*(pay)*}.

2.2 Applying Rule-Based Interceptors

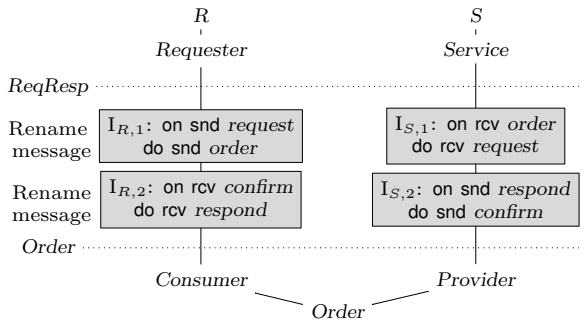


Figure 2: Evolving *ReqResp* to *Order*.

Figure 2 shows a simple, concrete example of our architecture, in which *Rebecca*'s agent R, enacting role *Requester*, sends a request message to *Steve*'s agent S, enacting role *Service*. S performs its function and responds. When the interceptor chains are empty, R and S interoperate using the *ReqResp* protocol (top dotted line).

Suppose the protocol designer, $\mathcal{P}$, determines that R is actually placing an order, and S is actually returning a confirmation. $\mathcal{P}$ then requires R and S must now interact with specialized protocol *Order* using messages *order* and *confirm* (bottom dotted line). Without needing to change either agent's implementation, $\mathcal{P}$ provides two interceptors for each agent's interceptor chain, evolving protocols from *ReqResp* to *Order*.

Figure 3 is a message sequence chart of the interaction, including both agents (solid lifelines) and all their interceptors

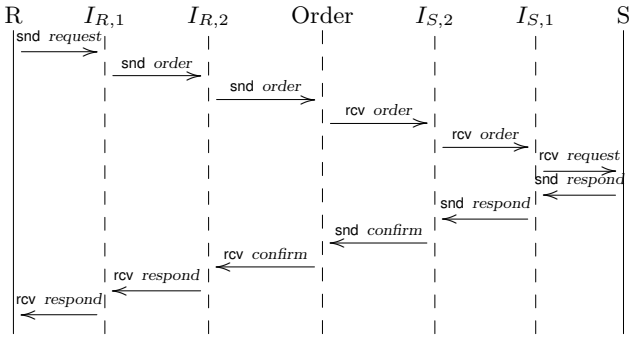


Figure 3: Detailed enactment of Figure 2.

and protocol (dashed lifelines). When R sends *request*, R's top interceptor ($I_{R,1}$) converts it to send *order*, which flows down to the protocol end of R's interceptor chain, and is sent over the protocol (Order) via the messaging infrastructure to S's interceptor chain. The *order* message flows up S's interceptor chain. Its top interceptor ($I_{S,1}$) converts it back to message *request*. S's *response* is converted to *confirm* in its bottom interceptor ($I_{S,2}$), and is sent over the protocol (Order) to R, whose bottom interceptor ($I_{R,2}$) converts it back to *respond*. The essential point of this example is both R and S use the original *ReqResp* protocol, even though messages of the *Order* protocol are what flow on the wire.

3. INTERCEPTORS AND REFACTORINGS

Given a set of agents that interoperate using a protocol, designers can incrementally *refactor* agent and protocol interactions to an evolved interaction, while preserving interoperability. *Interceptors* and *interceptor chains* mediate all message flow between an agent and a protocol using a reaction (event-based) architecture. A refactoring defines a set of coordinated and incremental changes to agents, interceptor chains, and the protocol. Interceptors and interceptor chains are the key elements that make refactorings possible.

3.1 Interceptor Chains

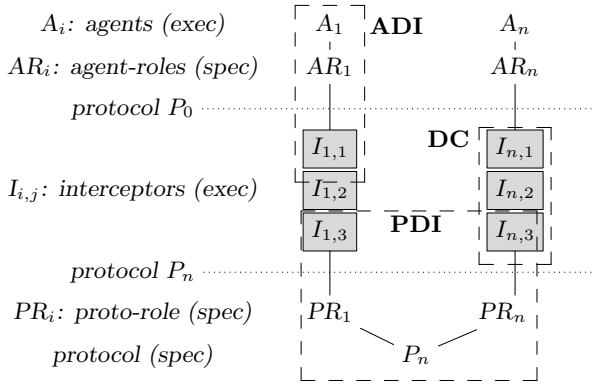


Figure 4: Interaction architecture. The dashed boxes signify the refactorings introduced above.

Figure 4 shows our *interaction architecture* consisting of agents (A_i), interceptor chains ($I_{i,*}$), interceptors ($I_{i,j}$), and protocols (P_k). Each agent, enacting role (or interface) AR_i of protocol P_0, communicates exclusively with the agent (top) end of its interceptor chain. An interceptor chain is an ordered list of interceptors (shaded boxes). The protocol (bottom) end of an interceptor chain, enacting role (or interface) PR_i of protocol P_n, communicates with the protocol end of other agents' interceptor chains. Agent implementations and interceptors are executable elements; roles and protocols are nonexecutable specifications.

In Figure 4, the top dotted line (P_0) separates the agent *implementation* and agent role (above) from the *middleware* of the interceptor chain and protocol (below). The bottom dotted line (P_n) separates the agent and interceptor chain *nodes* (above) from the protocol *interconnection* (below). Figures 2 and 5 are concrete instances of Figure 4.

Interceptor chains are the key to our approach because they enable the required designer independence: they insulate agents from protocol changes and insulate protocols from agent changes.

3.2 Interceptor Syntax and Semantics

Interceptor chains and interceptors are preprogrammed elements provided by the infrastructure, and require no designer implementation. We construct them using the following grammar

chain	:=	*role* : *interceptor**
interceptor	:=	*reaction* | *assertion*
reaction	:=	*onClause* (*ifClause*)$^?$ *doClause*;
onClause	:=	on *event*
ifClause	:=	if ϕ
doClause	:=	do *op* | do {*op**}
event	:=	rcv m | snd m to *role*
op	:=	rcv m | snd m to *role* | error | call *proc*
assertion	:=	kill *event*

where *role* is a role name, m is a message type, *proc* is a procedure name, ϕ is a Boolean expression, and BNF operators: $A_1|A_2$ (alternatives), A^* (zero or more repetitions), and $A^?$ (optional). The *doClause* is an ordered list of (1) receive operations (rcv m) that "call up" the chain, (2) send operations (snd m to role) that "call down" the chain, (3) throwing a run-time error (error), and (4) procedure calls to get or set interceptor chain data, or perform business functions. Assertions are design-time declarations and optionally perform run-time checks. The interceptor kill *event* asserts that *event* can never occur at a particular point in the chain at run time. It is used to propagate message deletions throughout the interaction.

At run time, an interceptor chain mediates all messages flowing in either direction between its agent and the protocol. The chain attempts to match each message to each interceptor, in order. A send message event starts at the agent (top) end of the chain, and "calls down" the chain toward the protocol end (bottom), passing over every interceptor in the chain in turn. A receive message event starts at the protocol end, and "calls up" the chain toward the agent end.

A send event (snd m to role) matches an (on snd m′ to role′) reaction, and a receive event (rcv m) matches an (on rcv m′) reaction, if the message types match ($m = m'$) and to roles match ($role = role'$). When the event matches both (1) the message and (2) the reaction's ifClause, if any, evaluates to

true in the current state, then the interceptor consumes the message and executes the list of operations in the doClause. Messages that reach the agent end of the chain are given to the agent; messages that reach the protocol end are given to the messaging infrastructure for delivery to the receiver.

3.3 Refactorings Formalized

A *refactoring* is a design-time construct, which encapsulates a coordinated and incremental set of interaction changes. For example, refactoring *Add Message* encapsulates all the interceptors for both the message sender and receiver. And refactoring *Add Middleman* encapsulates all the interaction changes for rerouting an existing message through a middleman, including all interceptors for the sender, middleman, and receiver.

A refactoring is a five-tuple that encapsulates one interaction-level change in high-level terms. As necessary, it can change all agents, all interceptor chains, and the protocol, applying an interrelated set of changes throughout, for example consistently renaming a message at both sender and receiver.

$R = \langle parameters, precondition, \Delta Agent, \Delta Chain, \Delta Protocol \rangle$

Given refactoring *parameters*, the *precondition* must be true at design time for the refactoring to be applicable. The refactoring applies changes to *Agents*, *Chains*, and *Protocols* at design time. Refactorings names are italicized, and each refactoring tuple is described with these common sections (omitting any empty sections)

- Parameters: input parameters to the refactoring.
- Preconditions: the preconditions that must be true before the refactoring can be applied.
- ΔAgent: changes to agents' implementations.
- ΔChain: changes to agents' interceptor chains. The notations $role.push_A : r$ and $role.push_P : r$ mean push interceptor r on to the chain's agent and protocol end, respectively.
- ΔProtocol: changes to the protocol.

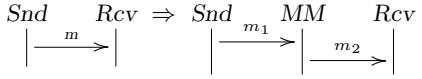
There are three groups of refactorings, each modifying a different set of elements, which we describe next.

3.4 Protocol Designer Independence

PDI refactorings modify the protocol, the protocol role, and the protocol end of interceptor chain. The protocol designer applies these refactorings to convert a group of interoperating agents from using protocol P_i to using protocol P_{i+1}. These refactorings isolate agents from protocol changes ($\Delta Agent = \emptyset$).

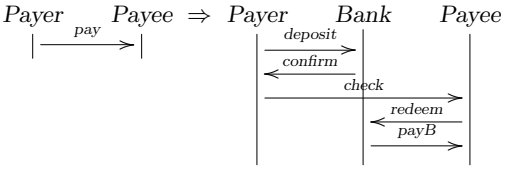
3.4.1 Add Middleman

This refactoring redirects a message to flow through a middleman agent. It replaces a single message m with a pair of sequential messages m_1 and m_2.



- Parameters:
  - m: existing message $Snd \rightarrow Rcv : [g]m$ means A
  - m_1: new message $Snd \rightarrow MM : [g_1]m_1$ means A_1
  - m_2: new message $MM \rightarrow Rcv : [g_2]m_2$ means A_2
- Preconditions:
  - Roles *Snd*, *MM*, and *Rcv* exist in the protocol.
  - m exists in the protocol.
  - m_1 and m_2 do not exist in the protocol.
  - $g_1 \vdash g$
  - All meanings in A, A_1, and A_2 exist
  - $A \subseteq A_1 \cup A_2$
- ΔChain: Add
  $Rcv.push_P$: on rcv m_2 do rcv m
  $MM.push_P$: on rcv m_1 do snd m_2 to Rcv
  $Snd.push_P$: on snd m to Rcv do snd m_1 to MM
- ΔProtocol: Delete m. Add m_1 and m_2.

This refactoring can be naturally extended to reroute through multiple middlemen, which is the variant we implement in our prototype. For example, it converts *pay* in *Pay* to *deposit*, *confirm*, *check*, *redeem*, and *payB* in *PayViaCheck*.



3.5 Agent Designer Independence

ADI refactorings modify the agent implementation, the agent role, and the agent end of the interceptor chain. The ADI refactorings isolate the protocol from agent changes ($\Delta Protocol = \emptyset$).

As examples, the agent designer can move functionality between the agent and its interceptor chain. Refactoring *Externalize Reaction* moves functionality out of the agent implementation and into the agent end of the interceptor chain, delegating agent functionality to a mechanistic reaction. Refactoring *Internalize Reaction* moves an interceptor off the agent end of interceptor chain and the agent designer merges that functionality into the agent's implementation.

Push Kill is another ADI refactoring that is described in Section 3.6.1.

3.6 Designer Collaboration

DC refactorings modify interceptors within a single interceptor chain. Protocol and agent designers can apply these refactorings to reorder, merge, or split interceptors within a chain to improve performance or to move interceptors toward one of the ends of the chain where they can be used in other refactorings. While refactorings in the other two groups isolate designers, these refactorings enable multiple designers to collaborate within an interceptor chain ($\Delta Agent = \emptyset = \Delta Protocol$).

3.6.1 Kill Message

This captures a set of three closely related refactorings—one ADI, one DC and one PDI—with similar parameters and preconditions, presented together for clarity. We iteratively apply these refactoring to move kill assertions around the interaction at design time (kill assertions are not typically present in chains at run time).

- Parameters:
  - Kill assertion kill snd m to Rcv
- Preconditions:
  - Protocol contains message m.
- ΔAgent: (ADI: *Push Kill*) Sending agent publicly declares it will never send m by publishing kill snd m onto its chain.
  - $Snd.push_A$: kill snd m to Rcv

- ΔChain: (DC: *Move Kill*) moves the kill up or down the chain. If a kill event matches the following on event, delete the reaction (left rule). If a kill event does not match the following onClause or doClause events, swap the interceptors (right rule). Similar rules apply for on rcv reactions, and two kill assertions commute (neither shown).

kill snd m on snd m do snd n ——— kill snd n	kill snd m on snd n do ... ——— on snd n do ... kill snd m

- ΔProtocol: (PDI: *Protocol Kill*) propagates the kill assertion from sender to receiver, deleting the message from the protocol.
  - $Snd.pop_P$: kill snd m to Rcv
  - $Rcv.push_P$: kill rcv m
  - Delete m from protocol

4. METHODOLOGY AND APPLICATION

In this section, we describe a methodology for selecting a sequence of refactorings. Then, we show how to evolve protocol *Pay* to protocol *PayViaCheck*, and how to support requirements evolution by propagating kill message assertions.

4.1 Methodology for Protocol Evolution

These steps guide the protocol designer in the evolution of an interaction from protocol P_i to P_{i+1}.

M1 Add or rename any roles so all new roles exist in the target protocol. Use *Add Role* (adds new role and empty interceptor chain) or *Map Role*.

M2 If any message is too coarse (one message with a larger-than-necessary set of meanings), split it into multiple, parallel messages using *Split Message*.

M3 Rename existing messages with *Rename Message*, and add new messages using *Add Message*.

M4 If any message needs to pass through one or more intermediary roles (common when adding new roles), reroute the messages using *Add Middleman*.

M5 If business function changes are required, add and delete procedure calls to the doClauses using *Add Procedure* and *Delete Procedure*.

M6 Combine parallel messages using *Merge Message*.

M7 Delete unneeded elements using *Remove Middleman* and *Remove Message*.

M8 Delete unneeded roles using *Remove Role*.

4.2 Evolve Pay to PayViaCheck

Algorithm 1 *Pay* Protocol

```
protocol Pay {
    role Payer; Payee;
    prop promise; pay;
    commitment
        C_pay = C(Payer, Payee, promise, pay);
    message
        Payer → Payee : promiseMsg
            means {promise, CREATE(C_pay)};
        Payer → Payee : [promise] payMsg
            means {pay};
}
```

Algorithm 2 *PayViaCheck* Protocol

```
 1: role Payer; Bank; Payee;
 2: prop acct; deposit; choose; check; redeem; payB;
 3: commitment
 4:     C_payB = C(Payer, Payee, deposit ∧ choose, check);
 5:     C_bank = C(Bank, Payer, deposit ∧ check ∧
    redeem, payB);
 6:     C_redeem = C(Payee, Bank, deposit ∧ check, redeem);
 7: message
 8: // Map promise to choose
 9:     Payer → Payee : [acct] chooseMsg
10:         means {choose, CREATE(C_payB)};
11: // Add Message
12:     Payer → Bank : openMsg means {open};
13:     Bank → Payer : [open] acctMsg
14:         means {CREATE(C_bank), CREATE(C_redeem)};
15: // Add Middleman to pay
16:     Payer → Bank : depositMsg means {deposit};
17:     Bank → Payer : [deposit] confirmMsg means {};
18:     Payer → Payee : [acct ∧ choose ∧ CREATE(C_payB)∧
19:         CREATE(C_bank) ∧ CREATE(C_redeem)]checkMsg;
20:         means {check};
21:     Payee → Bank : [choose ∧ check ∧ CREATE(C_payB)∧
22:         CREATE(C_bank) ∧ CREATE(C_redeem)]
23:         redeemMsg means {redeem};
24:     Bank → Payee : [acct ∧ check ∧ redeem]
25:         payBMsg means {payB};
```

We demonstrate how our refactorings convert *Pay* (Figure 1(a), Algorithm 1) to *PayViaCheck* (Figure 1(b), Algorithm 2), without requiring any agent implementations changes, using the following sequence of refactorings. Each step lists the methodology step number and the affected line numbers in Algorithm 2. Figure 5 shows the refactorings and interceptors.

1. *Add Role*: *Bank*. (Step M1, Line 1)
2. *Rename Message*: *promiseMsg* ↦ *chooseMsg*. The two protocols use different names for the same message. (Step M3, Lines 9-10)
3. *Add Message*: *open* and *acct*. During initialization (rcv *init*), *Payer* sends an *open* request, and *Bank* responds with an *acct* number. (Step M3, Lines 12-14)
4. *Add Middleman*: routes *pay* through multiple middlemen as *deposit*, *confirm*, *check*, *redeem*, and *payB*. When *Payer* pays, then deposit the money at *Bank*, wait for confirmation, and send *check* to *Payee*. *Payee* then redeems check at *Bank*, who responds with *payB*, which is converted back to *pay*. (Step M4, Lines 16-25)

4.3 Guard Propagation

Applying refactorings from all three groups enables agent and protocol designers to collaborate on interaction-wide changes. Assume we have a set of interoperating agents, using a payment protocol that supports multiple forms of payment, including Travelers Checks. If one particular (but not necessarily every) Purchaser decides to stop using Travelers Checks, it can publish that decision as a kill snd *payTC* assertion to its interceptor chain. Multiple refactorings propagate these change throughout the interaction as shown in Figure 6.

1. Applying *Push Kill*, *Purchaser* declares it never sends *payTC*.

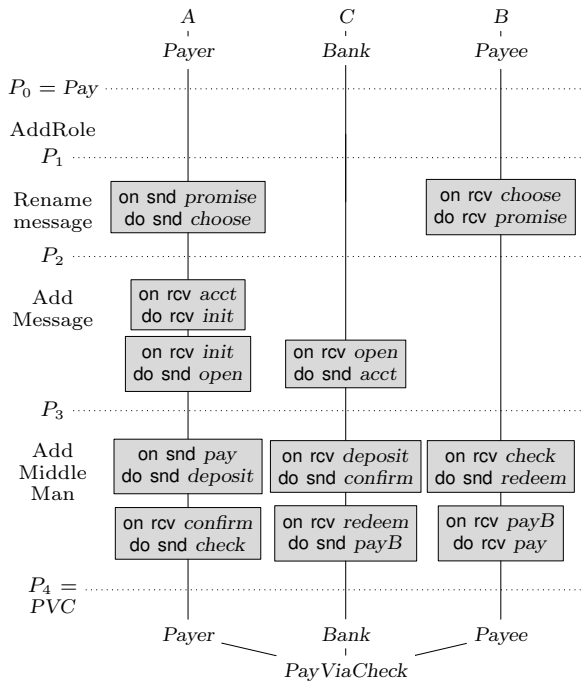


Figure 5: Evolution of *Pay* to *PayViaCheck* (PVC). Each shaded box is one interceptor.

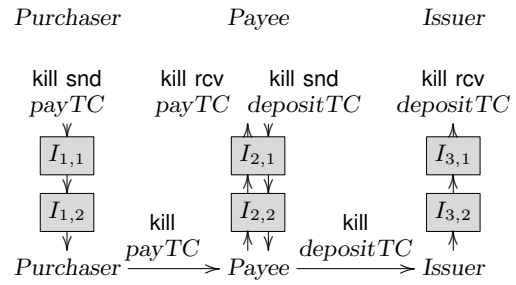


Figure 6: Removing unused message (false guard).

2. Repeated application of *Move Kill* moves this assertion down to *Purchaser*'s protocol end.
3. Applying *Protocol Kill*, the protocol designer propagates the kill assertion from sender to receiver and deletes *payTC* from the protocol.
4. Repeated application of *Move Kill* moves the kill assertion up to *Payee*'s agent end.
5. *Payee*'s agent designer pops the assertion off its agent end and internalizes it into its implementation.
6. Since *Payee* now never receives *payTC*, it realizes it never sends *depositTC*, and publishes kill snd *depositTC*, which propagates further.

The resulting protocol is different from the starting protocol, and is specialized for a particular set of agents. Overuse of these refactorings should be avoided to prevent an explosion of protocol variations. But when applied sensibly, these refactorings provide a natural means to incrementally evolve old interactions to handle changing requirements.

Designers have the option, but not the hard requirement, to move assertions as described above. Any assertion simply remains at its last location, where its optional run-time check will succeed on any acceptable enactment.

5. EVALUATION

We have prototyped these ideas using the JADE agent platform [3], with one JADE agent for each agent and one JADE "chain agent" for each interceptor chain. Each chain agent has its own thread of execution and message input queue, so it can send and receive messages without blocking.

First, at design time, a program builds the interceptor chains by applying refactorings. Second, at run-time initialization, each chain agent reads its interceptors. Third, as message events arrive from agents, the chain agent walks each message event up and down the interceptors in the chain.

An agent can be connected to multiple, different interceptor chains in different situations, enabling that single agent to simultaneously interact over different protocols.

Table 1 shows the number of refactorings and interceptors needed to evolve between protocols. Messages is the total number of messages in the final execution, including six initialization and six terminating, low-level messages not shown in Figure 1.

Table 1: Effort in evolving (refactorings and interceptors) and running (messages) sample protocols.

From	*To*	*Refactor*	*Intercept*	*Messages*
Pay	*PayViaMM*	2	5	19
Pay	*PayViaCheck*	4	11	24
Pay	*PayViaCredit*	5	12	24

Section 4.1's methodology offers guidance for designers, focusing on *Protocol Designer Independence* refactorings. However, Section 4.3 illustrates the key benefit that *Agent Designer Independence* and *Designer Collaboration* refactorings yield in interaction-wide changes. We claim our approach is easier than the traditional approach of manually changing agent implementation, because (1) predefined and verified refactorings are selected from a library, and (2) refactorings are at a much higher conceptual level than agent implementations.

Our interaction refactorings do not assist in refactoring business functions inside the agent's implementation.

Let reaction R be on snd m do snd n. If R is the only generator of snd n, then the *Move Kill* works correctly: if kill m, then kill n. But if R merges two existing messages, then it overkills snd n. While merging does not occur in the examples covered here, in general, it could. The current approach cannot adequately address such merging.

Refactorings can time-shift messages only within a limited range. The data values an interceptor passes in a message must come from previous messages or values stored in the interceptor chain. A message cannot be shifted earlier than the availability of all its parameters, and it cannot be shifted later than the next message that needs one of those values. This constraint required altering *PayViaCheck*'s deposit message, which originally required an up-front, one-time deposit. The refactored design can make a deposit just before sending each check. Without this change, the refactoring would not have been possible.

6. DISCUSSION

We identify and describe three forms of requirement evolution: *Protocol Designer Independence* (PDI), *Agent Designer Independence* (ADI), and *Designer Collaboration* (DC). Each focuses on different parts of an interaction, two provide designer isolation, and one enables designer collaboration. We describe refactorings for all three forms. Applying refactoring from all three forms, in concert, supports interaction-wide evolution. Interceptors and interceptor chains are the critical elements that enable refactorings.

We demonstrated refactorings to transform *Pay* into *PayViaCheck*, without changing agent implementations. We also demonstrated an agent voluntarily restricting its behavior (*payTC*) and propagating that change throughout the interaction.

This paper covers just a few refactorings, but we have defined a library with over 30 refactorings. A JADE prototype demonstrates basic interceptor chain functionality, refactorings automatically generating reactions, and agent interoperability after refactoring to a new protocol.

We adopt a reaction-based, interceptor chain architecture that is effective and yet simple enough to yield refactorings that are easy to understand and apply. Because interceptors are predefined and simple, we can define refactorings to mechanically evolve them. Mechanical evolution of general-purpose agent implementation is likely intractable.

Refactorings clearly communicate interaction changes. We mechanically transform refactorings into sets of interceptors. Interceptor chains can store important pieces of state (e.g., an agent's checking account information) and can make commitments on behalf of the agent (e.g., committing to redeem valid checks). In this case, the interceptor chain becomes a trustee of the agent, sometimes a necessity when unmodified agents participate in new protocols. However, it also raises autonomy concerns about the interceptor chain. Agents should be able to limit the trust and autonomy they grant to their interceptor chains.

6.1 Comparison to Design Patterns

Interceptor chains are a variation of the Chain of Responsibility (CoR) design pattern in Gamma et al. [7]. Vinoski [15] describes many uses of CoR. Servlet filters and filter chains [1] are a widely used example of CoR. But we know of no uses of CoR that support bidirectional flows. Nor do we know of a multi-interceptor design construct like ours.

Interceptor chains enable all the uses of servlet filters plus others. Servlet filter chains encourage servlet designers to consider moving function between a servlet and its filters (like ADI), and reordering filters within the chain (like DC). But servlet filters give no attention to the coordinated design of filters in different servlets (like PDI). Servlet chains do not support bidirectional flows.

Our interceptors are a bidirectional variant of the Bridge design pattern [7], also called a protocol bridge. Where a protocol bridge is a custom implementation, our small, preprogrammed interceptors are incrementally composed and refactored.

The Compatible Change pattern [5] describes a number of refactorings. The Service Refactoring pattern [5] applies only to service implementations, not interactions. Neither are mechanically applied.

6.2 Comparison to Agent Designs

In the traditional agent-only approach to agent design, evolution is subject to the vicious circle described in the introduction. Even when an agent designer decides to support a new protocol, implementation changes delay deployment. Agent designers can waste both time and effort implementing protocols that are never widely adopted. Using refactorings and interceptor chains breaks the vicious circle. Protocol designers dynamically update interceptor chains, essentially eliminating deployment delay. Agent designers spend time and effort implementing only after a protocol is widely adopted.

We demonstrated our approach by prototyping interaction evolution via interceptor chains in the popular agent platform JADE [3]. Jason, using the AgentSpeak language, is another popular agent platform. Our reaction's *onClause*, *ifClause*, and *doClause* are similar to an AgentSpeak plan's triggering event, context, and body, respectively. Both can send and receive "internal messages" (e.g., init) and call user-defined functions. Whereas beliefs, desires and intentions (BDI) are fundamental for autonomous functions in AgentSpeak, interceptor chains are not autonomous, so BDI does not apply. The primary problem these platforms have with evolution is that all computation occurs in designer-written agents, so all changes require designer effort, which can be expensive. It appears impossible in general to define mechanical refactorings that correctly and dynamically evolve JADE behaviors or AgentSpeak plans. We *can* define mechanical interceptor chain refactorings only because interceptor chains have a simple structure, reducing the designer's burden.

Agent UML (AUML) [10] informally describes agent interaction protocols (AIP), and promotes them as a means to define protocol interactions. However, Odell et al. [10] note that AIPs describe only one enabled sequence of message interactions. We formally define protocols as sets of guarded statements that capture all enabled message sequences. Guarded statements enable a relatively direct conversion [8] to modern model checkers such as MCMAS [9] and NuSMV [4]. Gerard and Singh [8] describe *protocol refinement*, but do not provide any guidance for constructing subprotocols. This paper describes both refactorings and a methodology to incrementally evolve interactions.

6.3 Comparison to Other Work

Quenum et al. [11] compose an agent from functional and interaction models via unification. They recreate (reconfigure) roles anew, in isolation, for each interaction model; we *incrementally evolve* (refactor) all agents in a protocol simultaneously.

Robinson and Purao [12] describe protocol invariants using OCL, which is based on predicate calculus and linear temporal logic, but they provide no rules to rewrite OCL statements. Because we use a simple reaction-based architecture, we can mechanically modify interceptor chains.

Fowler [6] refactors code; Wang et al. [16] refactor commitments; we refactor interactions.

Baldoni et al. [2] identify and discuss the important problem of patching agents to maintain interoperability. Seguel et al. [13] describe protocol adaptors (interceptor chains) to resequence messages between a pair of agent. We use a declarative approach in contrast with these two operational approaches. Neither approach supports as many protocol

changes as our refactorings, and they construct protocols rather than incrementally refactor them.

Serban and Minsky [14] describe an infrastructure for changing a distributed system while it is running. Their laws and controllers roughly correspond to our protocols and interceptor chains. They enable changes on running systems, where we consider changes only while the system is quiesced. Their users must manually design, write and test a completely new set of laws for a set of interacting agents; our designers expend less effort by incrementally evolving existing interceptor chains using a library of predefined refactorings. They provide no guidance on how to design and construct laws; we provide a methodology for refactoring interactions.

6.4 Comparison of Mechanistic Capabilities

In Table 2 we list various related approaches and whether they *mechanistically* support PDI, ADI, and DC style changes. PDI indicates the protocol can be changed by renaming messages, adding middlemen, and so on. ADI indicates the messages an agents sends or receives can be changed. DC indicates the internal organization of the interceptor chain equivalent can be changed, or is NA if no equivalent exists.

Approach	PDI	ADI	DC
Our Approach	Yes	Some	Yes
Chain of Responsibility (F) [7, 15]	No	Yes	No
Servlet Filter (F) [1]	No	Yes	No
Protocol Bridge [7]	No	No	No
Compatible Change [5]	No	No	No
JADE [3]	No	No	NA
Quenum et al. [11]	No	Yes	NA
OCL [12]	No	No	NA
Fowler [6]	No	Yes	NA
Wang et al. [16]	NA	NA	NA
Baldoni et al. [2]	Some	Yes	NA
Seguel et al. [13]	Some	No	No
Serban & Minsky [14]	No	No	No

Table 2: Compares representative agent and interaction programming approaches to mechanistically apply PDI, ADI, and DC changes. Some means partial support. (F) means unidirectional flow is from protocol to agent.

6.5 Future Directions

This work opens up interesting directions for future research. The current chain functionality is in a separate agent and requires minor changes to the way normal JADE agents send messages. Producing a modified JADE middleware that includes an interceptor chain component, whose contents can be changed at run time, supporting unmodified JADE programming patterns, would better enable evolution of service-oriented systems.

Replace the current extreme kill assertion with more flexible mechanisms that enable restricting a sender's, or relaxing a receiver's, private guard, capturing the "send less; receive more" intuition [2]. This will require careful tracking of valid events at every point throughout an interaction.

Provide formal verification of the soundness of our refactorings, possibly adapting techniques applied to protocols [8] as well as traditional model checkers [9, 4].

Acknowledgments

We are indebted to Jon Doyle, Anup Kalia, Pankaj Telang, Tao Xie, and the anonymous reviewers for helpful comments.

7. REFERENCES

[1] Java servlet specification, version 3.0. Sun Microsystems, Dec 2009.

[2] M. Baldoni, C. Baroglio, A. K. Chopra, N. Desai, V. Patti, M. P. Singh. Choice, interoperability, and conformance in interaction protocols and service choreographies. AAMAS, pages 843–850, 2009.

[3] F. Bellifemine, G. Caire, D. Greenwood. *Developing Multi-agent Systems with JADE*. England, 2007.

[4] E. M. Clarke, O. Grumberg, D. A. Peled. *Model Checking*. MIT Press, 1999.

[5] T. Erl. *SOA Design Patterns*. Prentice Hall, 2008.

[6] M. Fowler. *Refactoring*. Addison Wesley, 2000.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[8] S. N. Gerard, M. P. Singh. Formalizing and verifying protocol refinements. *TIST*, 2013. To appear; available at http://www.csc.ncsu.edu/faculty/mpsingh/papers.

[9] A. Lomuscio, H. Qu, F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. *CAV*, LNCS, pages 682–688, 2009.

[10] J. Odell, H. V. D. Parunak, B. Bauer. Extending UML for agents. In *AOIS*, 2000.

[11] J. Quenum, A. Slodzian, and S. Aknine. Automatic derivation of agent interaction model from generic interaction protocols. In *AOSE*, pages 193–229. 2004.

[12] W. N. Robinson, S. Purao. Specifying and monitoring interactions and commitments in open business processes. *IEEE Software*, 26(2):72–79, 2009.

[13] R. Seguel, R. Eshuis, P. Grefen. Constructing minimal protocol adaptors for service composition. In *WEWST*, pages 29–38, 2009.

[14] C. Serban, N. H. Minsky. In vivo evolution of policies that govern a distributed system. In *POLICY*, pages 134–141, 2009.

[15] S. Vinoski. Chain of responsibility. *IEEE Internet Computing*, 6(6):80–83, 2002.

[16] M. Wang, K. Ramamohanarao, J. Chen. Reasoning intra-dependency in commitments for robust scheduling. *AAMAS*, pages 953–960, 2009.