

# Hermes: Designing Flexible and Robust Agent Interactions

**Christopher Cheong**<sup>1</sup>

*School of Business Information Technology, RMIT University, Melbourne, Australia*

**Michael Winikoff**

*School of Computer Science and Information Technology, RMIT University, Melbourne, Australia*

## **Abstract:**

Although **intelligent agents** individually exhibit a number of characteristics, including social ability, **flexibility**, and **robustness**, which make them suitable to operate in complex, dynamic, and error-prone environments, these characteristics are not exhibited in multi-agent interactions. For instance, agent interactions are often not flexible or robust. This is due to the traditional **message-centric design** processes, notations, and methodologies currently used.

To address this issue, we have developed **Hermes**, a **goal-oriented design methodology** for agent interactions which is aimed at being pragmatic for practicing software engineers. Hermes focuses on interaction goals, i.e. goals of the interaction which the agents are attempting to achieve, and results in interactions that are more flexible and robust than message-centric approaches.

In this chapter, we present the design and implementation aspects of Hermes. This includes an explanation of the Hermes design processes, notations, and design artifacts, along with a detailed description of the implementation process which provides a mapping of design artifacts to goal-plan agent platforms, such as **Jadex**.

---

<sup>1</sup> Christopher Cheong carried out this research as part of his Ph.D. candidature in the School of Computer Science and Information Technology, RMIT University.

## INTRODUCTION

Our ever-evolving and technologically advanced world is a place that is complex, dynamic, and failure-prone. **Intelligent agents** are steadily accruing purchase as a technology which is intrinsically able to address the aforementioned real world issues (Jennings, 2001). Currently, intelligent agents are used in a range of real world applications spanning a number of different domains. These include telecommunication systems (Chaib-draa, 1995; Jennings 2001), process control (Sycara 1998; Jennings et al. 1998), air traffic control (Sycara, 1998), business process management (Jennings, 2001), logistics (Benfield et al., 2006), production scheduling (Munroe et al., 2006), and many more.

A key issue in developing and using agents is how to systematically analyse and design multi-agent systems. This issue has resulted in the development of the field of **Agent Oriented Software Engineering**. This field has seen the development of a number of **methodologies** which provide the developer with guidance, processes, and notations for the analysis and design of agent systems.

The systems in the previous examples all employ multiple agents as “there is no such thing as a single agent system” (Wooldridge, 2002). In such multi-agent systems, agent interactions are the crux of the matter, as the agents will need to interact in various ways in order to achieve their goals. Consequently, the design of agent *interactions* is a crucial part of a design methodology.

Current approaches to interaction design are **message-centric** as the design process is driven by messages that are exchanged during the interaction and is focused on the information passed within the messages. For example, in the **Prometheus** methodology (Padgham and Winikoff, 2004), as part of its interaction design process, the designers are advised to think about messages and alternatives. This is not restricted to Prometheus, but is also the norm in other methodologies such as **Gaia** (Zambonelli et al., 2004), **MaSE** (DeLoach et al., 2001) and **Tropos** (Bresciani et al., 2004).

Using current message-centric approaches to create interactions results in a number of problems. The main problem is that designs resulting from message-centric approaches tend to be overly, and sometimes unnecessarily, constrained. For example, using the interaction protocol of Figure 1, the interaction must begin with the Customer agent enquiring about the price of a laptop. It cannot, for example, enquire about the availability of a laptop first. Similarly, if a laptop is out of stock, the Vendor cannot proactively send a “Laptop Out of Stock” message to the Customer agent before or after replying with the price.

This lack of **flexibility** and **robustness** in interactions is problematic for intelligent agents. By following such limited designs, key intelligent agent characteristics, such as autonomy and proactivity, are greatly subdued and the fundamental concept of goal-orientation is ignored. Thus, current approaches to interaction design are not congruent with the agent paradigm.

More abstractly, the problem with message-centric approaches results from the general design process where the designer begins by creating a desirable but rigid message sequence and then “loosens it”, i.e. improves flexibility and robustness by adding alternatives. The problem with this is that the “default” result is an interaction that has not been sufficiently “loosened”, and is more constrained than it needs to be. A number of alternative approaches for specifying agent interactions have been explored. These alternative approaches avoid overly restricting interactions by starting with completely unconstrained interactions and then adding constraints so that the protocols are restricted and lead only to desirable interactions.

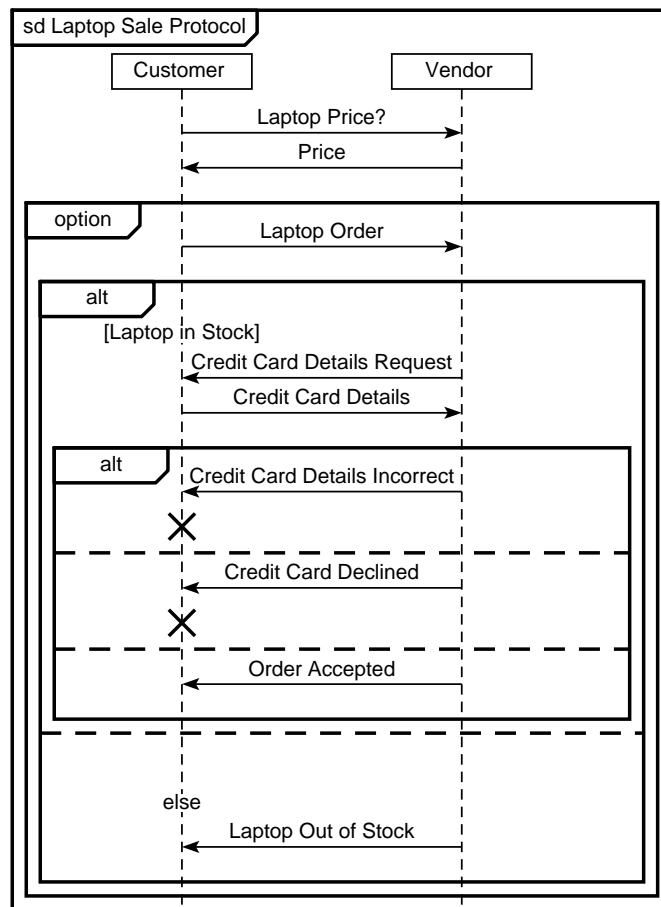


Figure 1: Message-Centric Protocol Example

In this paper we describe one such as alternative approach: the **Hermes** methodology. Hermes provides a way of designing agent interactions that results in **flexible** and **robust** interactions. A key design aim of Hermes was that it had to be *pragmatic*, that is, easily usable by designers. In particular, we wanted Hermes to include processes, notations, and guidance, not just a means for representing interactions. Furthermore, Hermes had to be usable by software engineers, not just agent researchers.

Hermes is intentionally limited to the design and implementation of agent *interactions* and not of entire agent systems as there are already many good methodologies for such in existence. Instead of competing with those methodologies, Hermes should be integrated with existing methodologies. Although not covered in this chapter, **Hermes** has been integrated with the **Prometheus** methodology (Cheong and Winikoff, 2006b). In the integrated methodology Hermes replaces the interaction design part of Prometheus, leaving the rest of the methodology unchanged.

The work, as currently presented, is intended for **closed agent systems**<sup>2</sup>. However, during the development of this methodology, the applicability of this work to **open systems**<sup>3</sup> was kept in mind.

<sup>2</sup> Systems in which the components are explicitly designed to interoperate.

Although not specifically designed for such systems, the methodology should be able to be adapted to work with open systems.

Thus the aim of this work, which has a software-engineering flavour, is to provide a practical *solution* to designing **flexible** and **robust** interactions between agents (primarily) in *closed* systems (where agents have more limited autonomy). By contrast, chapter 10, ‘Grounding Organizations into the Minds of Agents’, by Castelfranchi, has more of a philosophical flavour, and is concerned with posing *questions* about the nature of organizations, roles, agents, goals, and their relationships, including the process of goal adoption and delegation.

The remainder of this chapter is structured as follows. A background section reviews some background covering both traditional and alternative approaches to designing interactions. The following section presents the Hermes methodology, covering a process and design notations. We then present guidelines for implementing Hermes designs, and finally conclude and discuss future research directions.

## **BACKGROUND: INTERACTION DESIGN**

There are a number of ways in which agent interactions can be modeled and designed. The most obvious and simplest approach is to focus on information exchanged between interacting agents, i.e. the messages, and to specify and design interactions in terms of possible sequences of messages. This is indeed the approach that many current design methodologies use, and we refer to such approaches as “traditional” or “**message-centric**”.

Although simple and obvious, the problem with this approach is that it only captures the interaction at a superficial level. Focussing only on the communicative acts means that important information, such as the reason for uttering the communication, is not considered in the interaction design process. Furthermore, the interactions tend to be more restricted in terms of the range of possible interactions that are supported (termed “**flexibility**”), and in their ability to recover from failure (termed “**robustness**”). These shortcomings have motivated researchers to investigate a range of “alternative” approaches for agent interaction.

In addition to work that aims to make interactions (in **closed systems**) more flexible and robust, there is also work on designing (open) societies of agents. Rather than focusing on enabling the social ability of intelligent agents, this body of work is about defining societal-level mechanisms, such as norms, obligations and social laws, to provide rules of interactions for intelligent agents.

Two approaches to societal design of agent interactions are **Islander** and **OperA**. Islander (Vasconcelos et al., 2002; Esteva et al., 2002), an approach to **electronic institutions**, focuses on the macro-level (societal) aspects of multi-agents systems, rather than the micro-level (agent level). Electronic institutions are similar to their human counterparts in that they regulate what interactions can occur between agents. More specifically, an electronic institution defines a number of interaction properties, such as what interactions can occur, which agents can and cannot interact and under what circumstances these

---

<sup>3</sup> Systems in which the components are not explicitly designed to interoperate but are able to do so by adhering to published standards.

interactions can take place. In Islander, this is achieved by a global protocol which constrains the interactions between all components of the system.

Similarly to Islander, **OperA** (*Organizations per Agents*) (Dignum, 2004; Dignum and Dignum, 2003) takes a macro-level view of agent systems and focuses on agent societies rather than individual agents. The motivation for OperA is that most existing agent-oriented methodologies design agents from the individual agent perspective, however, a wider perspective, such as a societal-level one, is required to design agent societies as a society, i.e. not just a collection of individual agents interacting together. Furthermore, some societal-level goals cannot be captured as a collection of individual agent goals. Capturing societal-level goals allows for the analysis of societal characteristics, which is a motivation for OperA. OperA is a model and a design methodology for creating such agent societies.

The remainder of this section describes the traditional message-centric approach, and then surveys a number of alternative approaches.

### **Traditional Message-Centric Approaches to Agent Interactions**

Agent interactions have traditionally been specified in terms of **interaction protocols** expressed in notations which focus on the message exchanges between the agents. Common notations for expressing such agent interactions are **Agent UML (AUML)** (Odell et al., 2000; Huget and Odell, 2004), **Petri nets** (Reisig, 1985), and finite state machines. AUML sequence diagrams (Huget et al., 2003) are quite often used to specify agent interactions, and has been adopted by methodologies, such as **Gaia**, **MaSE**, **Prometheus**, and **Tropos**, and is commonly used. It should be noted that the AUML sequence diagram has developed from its original version (Bauer et al., 2000 & 2001) to a more recent version (Huget et al., 2003; Huget and Odell, 2004) which is influenced by **UML 2.0**.

In relation to our goal of designing flexible and robust agent interactions we observe that although the AUML sequence diagram is frequently used to represent agent interactions, it has a number of problems associated with it. In theory, it is possible to have an unlimited number of alternative message sequences; thus, it is possible, in theory, to create a very flexible and robust design in which all alternatives are catered for. However, in practice, this is impractical and cumbersome, and would result in an AUML sequence diagram which is difficult to read, understand, and manage. This practical limitation is attributed to the fact that the notation for AUML sequence diagrams focuses on the sequences of messages exchanged between agents in the interactions.

Furthermore, although AUML itself is a notation and not a design process, the design processes often employed with AUML sequence diagrams lead to designs that exhibit limited flexibility and robustness. For example, when developing interaction protocols using **Prometheus**, the interaction designer is focused on identifying alternative out-going messages in response to incoming messages and, thus, does not have the autonomy and proactivity of intelligent agents foremost in mind when designing these interactions.

These processes typically start with a desired set of message exchanges between agents in an interaction and then are generalized by the addition of alternative message sequences. That is, the design begins with a very restricted interaction and then proceeds to “loosen” or “relax” it by adding alternatives. This tends to result in designs that have a limited number of alternatives as it is a cumbersome process to add many alternatives and it is also impractical to add many alternatives using the AUML sequence diagram notation. By contrast, Hermes is somewhat better able to capture alternatives, and its design process leads

the designer to explicitly consider failure points, resulting in a more flexible and robust design (see the evaluation discussed in the conclusion section of this chapter).

Thus, the **AUML** sequence diagram notation, in concert with **message-centric design** processes, leads to designs that have limited alternatives. As the agents are bound to follow the designed interactions, they are restricted to the limited number of alternatives that the interaction designers have allowed.

For example, using the interaction protocol of figure 1, the interaction must begin with the Customer agent enquiring about the price of a laptop. It cannot, for example, enquire about the availability of a laptop first. This is inflexible since the Customer may be more concerned about availability (e.g. if the price is standard and they need the machine quickly). Similarly, if a laptop is out of stock, the Vendor cannot proactively send a “Laptop Out of Stock” message to the Customer agent before or after replying with the price. Furthermore, this protocol does not allow for an alternative credit card to be used, should the provided card be declined.

A better approach to interaction design is to start at the opposite end of the spectrum. That is, to begin with a completely unconstrained interaction and then proceed by adding constraints to restrict the agents so that they produce desirable interactions. Because the design is by default unconstrained, rather than constrained, this tends to lead to a greater number of alternatives, resulting in greater flexibility and robustness in interactions. Such alternative approaches to the traditional message-centric approach are discussed in the following section.

## **Alternative Approaches to Traditional Agent Interactions**

There are various alternative approaches to the traditional message-centric interaction design. These alternative approaches, in contrast to message-centric approaches, diverge from focusing on the messages to design the interaction. Instead, they focus on various other elements of the interaction, such as social commitments or the states of the interaction which guide the agents to communicate (i.e. exchange messages). Thus, these alternative approaches are at a higher level of abstraction than message-centric approaches.

Although the end product is still agent interactions in which agents communicate through exchanges of messages, designing these interactions at a higher level of abstraction has a number of advantages. The foremost of which is that valid message sequences *emerge* from the interaction in a less constrained manner, which increases the **flexibility** of the interaction. This is quite different to message-centric interaction design in which, as explained in Section 2.1, valid message sequences must be *explicitly specified* and are often too constrained.

Alternatives to message-centric design includes **commitment-** and **landmark-**based approaches, along with a number of other alternative approaches. In commitment-based interactions, agents are guided by social commitments to communicate and progress through the interactions.

There are a number of approaches based on the notion of **social commitments** (Singh, 1991; Castelfranchi, 1995). Also see chapter 11 ‘Modeling Interactions via Commitments and Expectations’ by Torroni *et al.* which also considers using *expectations* as an alternative approach to defining the semantics of interactions. One reason for the popularity of social commitment-based approaches is that social commitments are *verifiable*. That is, social commitments are independent from an agent's internal

structure and mental states and are observable by other agents. These are two important properties, as the first allows the social commitment to be utilized by heterogeneous agents and the second allows all agents involved in the interaction to determine if a commitment has been violated or not.

In **commitment machines** (Yolum and Singh, 2002 & 2004), a (social) commitment between two agents represents one agent's responsibility to bring about a certain condition for the other agent. There are two types of commitments in commitment machines: base-level and conditional commitments. A base-level commitment is denoted as  $C(x, y, p)$ , which states that a debtor,  $x$  must bring about a condition,  $p$ , for creditor,  $y$ . A conditional commitment is denoted as  $CC(x, y, p, q)$  and states that if a condition  $p$  is brought about, then debtor  $x$  will be committed to creditor  $y$  to bring about condition  $q$ .

For example, consider an e-commerce example based on a simplified version of the **NetBill** protocol, taken from (Yolum and Singh, 2002), in which a *customer* is attempting to purchase a product from a *vendor*. One possible action is for the vendor to send a quote to the customer. This action has the effect of creating two conditional commitments: the first is an offer that, should the customer accept the offer, the vendor will then be committed to delivering the product<sup>4</sup>; the second commitment is for the vendor to provide a receipt conditional on the customer having paid. Formally these commitments are  $CC(\text{vendor}, \text{customer}, \text{agree}, \text{productDelivered})$  and  $CC(\text{vendor}, \text{customer}, \text{paid}, \text{receiptSent})$  where *agree* is itself the commitment  $CC(\text{customer}, \text{vendor}, \text{productDelivered}, \text{paid})$ .

The interaction is driven by the (base-level) commitments of the agents. In this example, once the customer accepts the offer from the vendor, the first commitment above becomes a base-level commitment to ensure that products are delivered. Once this is done the customer's agreement becomes the base-level commitment to pay, and once payment is received the vendor's second commitment becomes a base-level commitment to send a receipt.

A key point in the approach is that this particular sequence is only one of many possible sequences. For example, another, equally valid, interaction begins with the vendor shipping the goods (this may make good sense if the goods are "zero cost", e.g. software) and the customer may then decide whether to pay for the goods. Another interaction sequence begins with the customer accepting (i.e. committing to pay for goods should they be provided).

The work of Flores and Kremer (2004b, 2004a), is another approach based on social commitments, however, their notion of commitment is slightly different to that of commitment machines. They view a social commitment as an agreement between two agents in which one agent is responsible for the *performance* of a certain action for the other agent. Note that the debtor does not necessarily have to perform the action itself; it is only responsible for that action being performed, whether it performs it itself or employs another agent to perform it. As with commitment machines, the agents progress through the interaction through the attainment, manipulation, and discharge of commitments.

A third commitment-based approach to agent interactions is the work of Fornara and Colombetti (2002, 2003). As with the previous approaches, the social commitments are utilized to drive the interaction. However, in this body of work, the commitments are defined as an abstract data type, the *commitment class*, which can be instantiated into a *commitment object*. The commitment abstract data type consists of

---

<sup>4</sup> More precisely, to reach an interaction state where the product has been delivered.

a number of fields (such as debtor, creditor, state, content, and condition) which describe the properties of a commitment and a number of methods (such as make, cancel, reject) which are used to manipulate it.

Chapter 14, 'Specifying Artificial Institutions in the Event Calculus' by Fornara and Colombetti provides an analysis of commitments and their life-cycle using the event calculus, and relates commitments to the larger picture of institutions.

Although a social commitment approach is suitable for creating more flexible and robust interactions than current message-centric approaches, it has a number of disadvantages. Commitment-based approaches have only been applied to a few small examples and it is not clear whether they are applicable to larger or more realistic interactions. Additionally, it is unclear what software tool support exists for the facilitation of creating interactions based on commitment machines.

Another disadvantage is the lack of mature design processes for creating agent interactions using **social commitments**. That is, given a particular interaction, it is not obvious what commitments are required to create a commitment-based interaction. The work in (Yolum, 2005) describes a number of protocol conditions to be checked and provides algorithms to check these conditions. A methodology for the design of **commitment machines** has been recently presented (Winikoff, 2006) (along with a process for mapping commitment machine designs to a collection of plans (Winikoff, 2007)). However, this is only an initial methodology and it has not been applied to a wide range of examples. Furthermore, this methodology begins interaction design with a Prometheus-style scenario, which is a sequence of ordered steps. This tends to result in designs that are constrained and do not exploit well the flexibility and robustness that commitment machines are able to achieve.

Thus, although promising, designing flexible and robust interactions using social-commitments does not yet seem to be a usable and pragmatic approach for practicing software engineers. In contrast, this work aims to provide a pragmatic methodology for the design of flexible and robust agent interactions. This methodology has been applied to a larger range of interaction than the initial commitment machine methodology presented in (Winikoff, 2006).

In the **landmark**-based approach (Kumar et al., 2002a; Kumar et al., 2002b; Kumar and Cohen, 2004), a *landmark* represents a particular state of affairs and agent interactions are represented by a set of partially ordered landmarks, which can be initial (the start of the interaction), essential intermediate landmarks, optional intermediate landmarks, or final landmarks where the interaction terminates. Agents navigate through the landmarks to reach a final desired landmark, that is, a desired state of affairs, proceeding from one landmark to another by communicating with one another. The landmarks and their partial ordering can be depicted as a graph.

In this work the states of affairs are more important than the actions (i.e. communicative acts) that bring them about. Thus, as with the commitment-based approaches, the message sequences are not explicitly defined, but rather, they emerge as the agents communicate in an attempt to reach a final desired state of affairs.

The landmarks approach is theoretical in nature and has a heavy reliance on expertise in modal and temporal logics, which practicing software engineers typically will not have. Although an

implementation, *STAPLE*, has been mentioned, there have been no further details apart from the publication of two posters (Kumar et al., 2002b; Kumar and Cohen, 2004).

More closely related to the research in this chapter is the goal-plan approach of Hutchison and Winikoff (2002), in which interactions are realized using the plans and goals of **Belief-Desire-Intention (BDI)** agents. The work proposed a process to translate a message-centric protocol to a set of goals and plans. The work can be seen as a predecessor to this research. However, although a design process was outlined, it was not detailed and there is no mapping from design to implementation. Further to lacking a clear design process, as with the aforementioned approaches, the goal-plan approach has not been integrated with any existing full agent system design methodologies.

Although this section describes alternative approaches to traditional message-centric design, there is not much discussion about how to *design* agent interactions in the presented approaches. The approaches focus on novel ways in which more flexible and robust agent interactions can be represented and achieved, but as yet, they do not focus on how the interactions can be *designed* using these novel approaches. In the previously described approaches, the lack of design processes and methodologies is a recurring disadvantage and limitation. In fact, this lack of design processes is one of the key motivation for the research described in this document.

## **HERMES DESIGN PROCESS**

In this section, we present the design aspect of the Hermes methodology. The contributions of this section are:

- A design process that guides the designer to create **goal-oriented** interactions from an initial high level description of an interaction through to a design which can be implemented on goal-plan agent platforms, including steps for identifying and handling failure.
- Failure handling mechanisms which increase the **flexibility** and **robustness** of the goal-oriented interactions.
- Notation<sup>5</sup> for capturing and modeling key goal-oriented design artifacts.

We begin with an overview of the Hermes methodology, and then progress through the design process in subsequent sections. Note that Hermes is not a full design methodology. It focuses solely on aspects relating to designing interactions, and is missing other aspects such as identifying agent types, defining the internals of agents, or delineating the boundary between the agents and their environment.

### **Methodology Overview**

An overview of the Hermes methodology is shown in Figure 2. The methodology is divided into three phases which are performed in an incremental iterative manner.

---

<sup>5</sup> There are some similarities between the Hermes notations, especially action maps, and UML. Space precludes a detailed comparison, see (Cheong, 2008) for details.

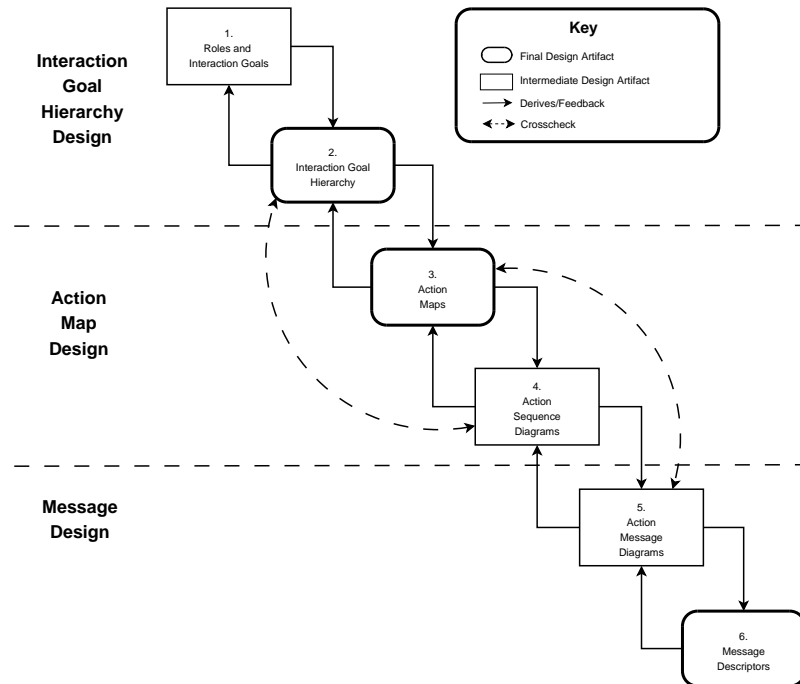


Figure 2: Hermes Methodology Overview Diagram

The first two steps fall into the first phase, the *Interaction Goal Hierarchy Design* phase, in which the designer is focused on the overall design of the interaction. The designer is concerned with *what* the interaction is to achieve and *who* (i.e. which roles) are involved in the interaction. Thus, Hermes begins by identifying roles and interaction goals as they are required before one can develop actions in the second phase. The identified roles and interaction goals are simply captured in a list. Afterwards, in the second step, these interaction goals are organized into an **Interaction Goal Hierarchy** (see the next section) , which is the final<sup>6</sup> artifact produced by this phase.

The second phase, the *Action Map Design* phase, requires the designer to think about *how* the roles involved can achieve the interaction goals identified in the previous phase. As such, actions which the interacting roles will need to carry out are identified in step 3 and are organized into appropriate execution sequences. In step 4, these execution sequences are checked. The final artifacts produced by this phase are the **Action Maps** (resulting from step 3), which define possible sequences of actions executed by the roles to achieve the interaction goals. There may also be intermediate **Action Sequence Diagram** artifacts (resulting from step 4), which are used to ensure that the sequences in the action maps are sufficient to allow the roles to achieve the interaction.

In the last phase, the *Message Design* phase, the designer's attention shifts from actions to communications between the roles, i.e. messages, as they are required to complete the interaction definition. Step 5 requires the designer to identify where messages are required to be exchanged between

<sup>6</sup> A *final* design artifact is defined as a non-intermediate artifact that is retained for design documentation purposes. In some steps of Hermes, *intermediate* artifacts are created to either provide a logical path which will guide the designer from one step to another or to allow the designer to check the created design. These artifacts are not intended to be retained as design documentation.

roles, while step 6 calls for the designer to define what information the messages will contain. The final artifacts from this phase are the message definitions, which will vary depending on whether the designer is using platform-specific message types or standards, such as **KQML**, **FIPA** or **SOAP**. The message definitions are recorded in *message descriptors*.

The following sections explain each of the phases and steps of the design process in detail and provides an example of how a design is created in Hermes.

## Interaction Goal Hierarchy

The first step in creating the interaction goal hierarchy is to determine the roles involved in the interaction and the **interaction goals** which they are attempting to achieve. Consider an agent type, *Academic Agent*. This agent can take on a number of different roles in different interactions. For example, in an academic paper reviewing interaction, the *Academic Agent* can undertake any of the following roles: *Author*, *Reviewer*, *Editor*, etc.

Therefore, a role usually represents a subset of what an agent can do and one agent is able to assume other roles in other interactions. Although possible, it is not usual for one agent to assume multiple roles in the same interaction. In fact, there may be rules preventing an agent from undertaking multiple roles in the same interactions. For example, in the aforementioned academic paper reviewing interaction, an agent cannot play both the roles of *Reviewer* and *Author* on the same paper.

The second step of developing the **interaction goal hierarchy** is to refine and organize the interaction goals identified in the previous step. Where possible, the interaction goals are broken down into smaller sub-interaction goals and are organized into a hierarchy. The hierarchy should only have one interaction goal at its apex, which captures the overall goal of the entire interaction.

As an example, consider a scenario in which four agent roles, *Sales Assistant*, *Customer Relations*, *Delivery Manager*, and *Stock Manager*, are interacting to fulfil an *Order Book* request in an online store<sup>7</sup>. The *Sales Assistant* is the main interface to the customer who places the online order whilst the *Customer Relations* maintains customer details, such as customer records. The *Delivery Manager* fulfils deliveries to the customer and the *Stock Manager* keeps track of inventory levels. In general, the steps of the interaction involve retrieving the customer's details, accepting payment, and shipping the book to the customer. Logs and records will also need to be updated as required.

As the top-most goal of the interaction goal hierarchy is usually the most abstract goal and is meant to capture the overall intent of the interaction, *Order Book* is placed at the apex of the hierarchy.

To continue developing the interaction goal hierarchy, more interaction goals are identified from the textual description of the interaction and are placed into the interaction goal hierarchy using decomposition relationships, that is, interaction goals are placed into parent-child relationships. Some of these interaction goals that will need to be added will be obvious to the designer. The remaining goals can be identified using either a top-down or bottom-up approach (or a mixture). In the top-down approach, the designer analyzes each existing interaction goal and determines if it can be decomposed

---

<sup>7</sup> This scenario is based on the book store design of (Padgham and Winikoff, 2004)

into smaller, more concrete goals. Decomposition should stop before producing goals that do not require interaction between roles (i.e. that can be achieved by a single role). Taking a bottom-up approach requires the designer to identify and aggregate bottom-level, concrete interaction goals into abstract ones and progress up the interaction goal hierarchy.

An example interaction goal hierarchy is shown in Figure 3 in which the undirected lines denote parent-child or sub-goal relationships. The lines from *Order Book* to *Retrieve Details* and *Order Book* to *Process Order* state that for the *Order Book* interaction goal to be achieved, the *Retrieve Details* and *Process Order* interaction goals must be achieved. Furthermore, the *Retrieve Details* and *Process Order* interaction goals are achieved when their sub-goal are achieved.

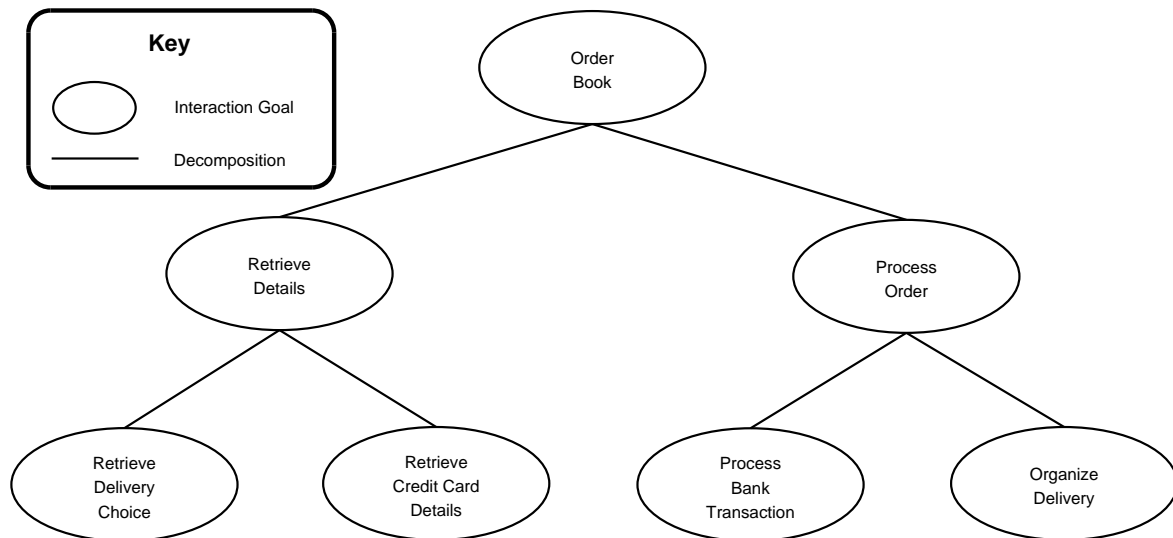


Figure 3: Intermediate Interaction Goal Hierarchy

The interaction goal hierarchy is, in actuality, a goal-tree, similar to those used in methodologies such as **MaSE** (DeLoach et al., 2001; DeLoach 2006) or **Prometheus** (Padgham and Winikoff, 2004). Leaves in the tree (i.e. goals with no children) are termed *atomic* interaction goals. Achieving the other interaction goals in the hierarchy, which are named *composite* interaction goals, is done by achieving the atomic interaction goals.

Once the designer has settled on an appropriate interaction goal hierarchy, temporal dependencies<sup>8</sup> (depicted as directed lines in Figure 4) are added. The temporal dependencies provide an effective way for the designer to place *constraints* on the sequence of the interaction and, thus, restrict the order in which certain interaction goals can be achieved. For example, in Figure 4, the directed line between *Retrieve Details* and *Process Order* depicts a temporal dependency between the two interaction goals and states that the *Retrieve Details* interaction goal must be achieved (successfully) before the *Process Order* interaction goal can start.

<sup>8</sup> These dependencies are different to causalities: they state that one interaction goal cannot begin until a preceding interaction goal has completed. A causality would state that a given interaction goal causes another interaction goal to be achieved. This distinction is more apparent when the interaction goal hierarchy does not specify a sequence of interaction goals.

While temporal constraints are useful to restrict certain undesirable sequences of interaction goal achievement from occurring, they should be used loosely as the more temporal constraints are used, the less flexible the interaction. For example, the particular design shown in Figure 4 is a strongly constrained design, however, alternative designs could, for instance, retrieve the delivery choice and credit card details simultaneously, thus, relaxing some of the temporal constraints.

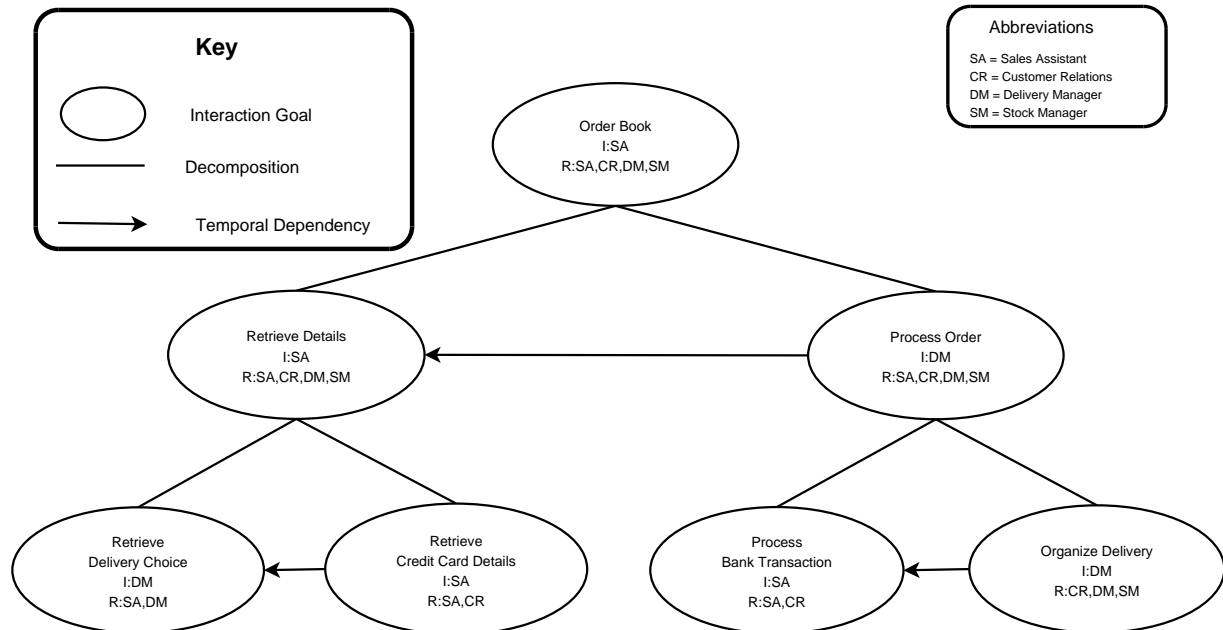


Figure 4: Final Interaction Goal Hierarchy

The placement of temporal dependencies will depend on the designer and the interaction itself. In general, they are placed to ensure that interaction goals are achieved in a sensible sequence. For example, it does not make sense for the roles to achieve the *Process Order* interaction goal before achieving the *Retrieve Details* interaction goal.

As part of developing the interaction goal hierarchy, the designer should also assign to each interaction goal the roles that are involved in that interaction goal. Typically, all roles involved in the interaction will be involved in each interaction goal, however, there may be situation in which only a subset of the roles are involved in particular interaction goals. The roles involved are shown in Figure 4 as (e.g.) *R: SA, CR, DM, SM* (short for Roles: Sales Assistant, Customer Relations, Delivery Manager, and Stock Manager).

The designer must also identify for each interaction goal which role is the *initiator*. The initiator can be one of the roles involved in the interaction goal, or the symbol  $\uparrow$  indicating that the initiator is *inherited*, i.e. the initiator for the interaction goal is the same as the initiator of its parent. In Figure 4 there are no inherited initiators, however, in other interactions it might be desirable to be able to state that the agent that began the interaction is responsible for initiating a particular interaction goal (i.e. an inherited initiator) (Cheong and Winikoff, 2006a). This provides more flexibility in the design of interactions.

In some cases it may not be clear which role should initiate a given interaction goal. In such cases an inherited initiator can be used initially and later on, in the action map design, the designer can use the initial actions (see the section on developing initial action maps below) to determine which role should be

the initiator. However, this approach is only suitable for atomic interaction goals since only atomic interaction goals have associated action maps.

The **interaction goal hierarchy** provides an overview of the interaction and depicts what the interacting roles need to achieve in order to achieve the interaction. Up to this point, only the common and coordination aspects of the interaction have been designed. The next step in developing a Hermes interaction is to consider how the interaction will be realised. This is done by creating an action map for each atomic interaction goal. Each action map shows *how* its corresponding interaction goal is to be achieved. The development process of action maps is described in the following section.

## Action Maps

For ease of explanation, the action map development process is described in three<sup>9</sup> distinct steps, however, it is not intended that designers rigidly follow these steps.

The steps are as follows:

1. Develop the initial action maps.
2. Add data to the action maps and consider data flow issues.
3. Generalize the action maps.

Typically, one action map is created for each atomic interaction goal, however, due to space limitations, we explain the development of only one action map in this document.

### Develop Initial Action Maps

The initial development of action maps is broken down into three steps:

- A. Identify actions and assign them to roles involved in the interaction;
- B. Establish action sequences by use of causality links; and
- C. Identify the type of each action.

The first step in developing an action map is to identify actions and assign them to the roles involved by placing them into the appropriate role's swim lane. To identify what actions are required, the designer will need to consider the abilities of the relevant roles and what interaction goals they have to achieve.

For example, consider the *Organize Delivery* interaction goal in Figure 4. All four roles are involved, and, thus, all four are present in the corresponding action map, shown in Figure 5. To achieve the *Organize Delivery* interaction goal, a number of actions, such as *Place Delivery Request*, and *Log Outgoing Delivery* will need to be identified and allocated to roles as in Figure 5.

---

<sup>9</sup> In an earlier presentation we had four steps, here we have merged steps three and four together.

## Organize Delivery (Initial without data stores)

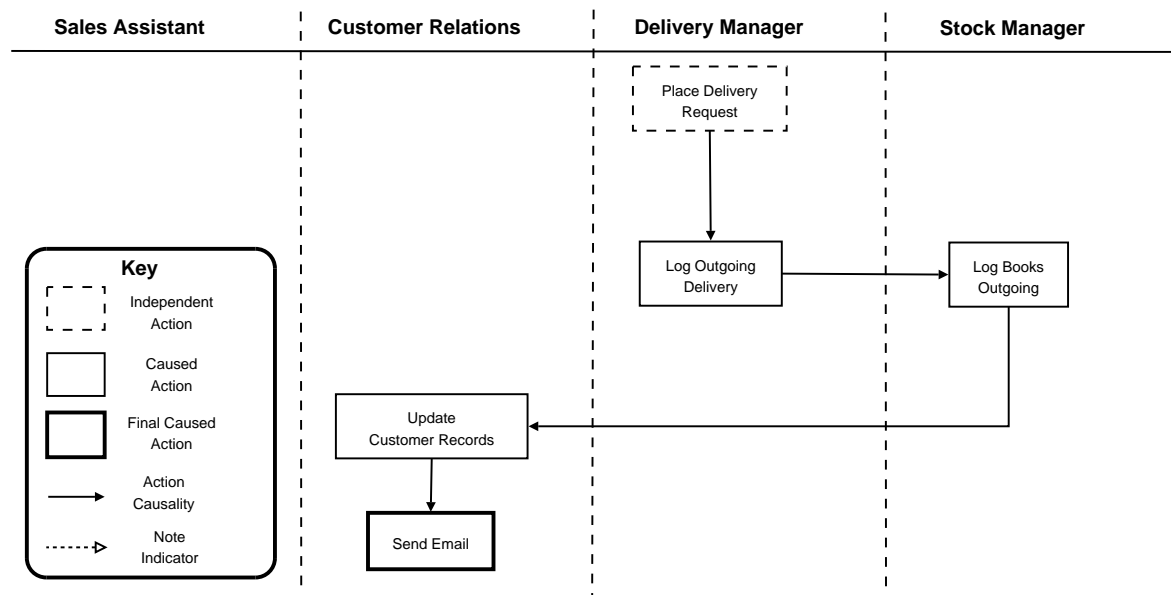


Figure 5: Initial Action Map

Once actions have been identified and assigned to the involved roles, the action execution order must be established, which is achieved by placing causality links between the actions. The causality links impose temporal restrictions and indicate the *action flow* of the action map, i.e. which actions can be attempted after an action has been executed. **Causality links**<sup>10</sup> are not necessarily inter-agent; they can also be intra-agent. Furthermore, causality links are able to be labeled with conditions or states. This is useful to clarify the causality paths on the action map.

Where to place causality links will depend on the designer and the interaction, and is usually common sense. For example, in the online book store, the *Delivery Manager* firstly places the delivery request and then logs the outgoing delivery, hence, a causality link is placed between the two actions. To ensure that all causality links have been identified, the designer should check each action against all other actions and ensure that the established sequence is sensible. This will involve checking that dependencies are correct (e.g. *Send Email* cannot occur until the records have been updated) and that all actions are reachable (i.e. all actions are connected by causality links).

The last part of the initial development of the action maps is to determine the **action type** of each action. The action types are needed to indicate which actions start and terminate the action maps, because it is necessary to allow for multiple start and end points. The different action types are:

<sup>10</sup> To clarify the difference between causality and dependency, consider a situation in which there are three actions: *Action A*, *Action B*, and *Action C*. *Action A* causes *Action C* and *Action B* also causes *Action C* (i.e.  $A \rightarrow C \leftarrow B$ ). In this case, *Action C* is triggered when either *Action A* or *Action B* complete because causalities are used. However, if arrows are viewed as dependencies, then *Action C* can only occur after *both Action A* and *Action B* have completed (as *Action C* depends on both).

- *Independent Actions*, denoted as a rectangle with dashed border, which can start without being triggered by another action (although they also can be triggered). Typically independent actions are used as entry points to the action map.
- *Caused Actions* which *must* be triggered by another action and are denoted as a rectangle with solid line.
- *Final Caused Actions*, denoted by a rectangle with thick solid line, which are caused actions which terminate an interaction goal, i.e. once done, no further actions will be executed in that action map.
- *Final Independent Actions*, denoted by a rectangle with a thick dashed line, which are independent actions which terminate an interaction goal. This action type is rarely used since it corresponds to a situation where the action is both the initial and final action, i.e. it is the only action used.

At the end of this step, a rudimentary **action map** is created, for example see Figure 5. The consequent steps will refine it into a more **flexible**, **robust** and complete action map.

### Adding Data to Action Maps

This step involves identifying and adding data stores to the action maps. This is important as particular actions will require appropriate data. The designer must also ensure that data which the roles require is accessible. To identify the necessary data stores, the designer analyzes the actions carefully and considers what data is required for the actions to execute successfully. It is also useful to determine what data needs to be passed from one action to another. Once the data has been identified, data stores are placed in the swim lane of the role to which they belong. Note that only relevant data stores are displayed on an action map; not all the data stores that a role contains. This avoids unnecessarily cluttering the action maps.

For example, in Figure 6, the *Customer Relations* role will need to store customer records somewhere. This is captured by its *Customer DB* data store. Similarly, the *Delivery manager* will need to keep track of customer orders and the *Stock Manager* will need to manage inventory. These are represented by their *Customer Orders* and *Stock DB* data stores respectively.

Simply adding data stores is not sufficient, the designer must ensure that actions which read and write data have *direct* access to the data stores. The designer must also ensure that all actions will have access to data even if the data store resides in another role. This may mean that required data is read from a data store and is passed along through multiple actions to reach a particular action that requires the data. In order to ensure all these, the designer should consider for each action what data is needed, where the data will be obtained from, and where the data will (finally) end up.

For example, the *Store Manager's Log Books Outgoing* action might need details from the customer records located in the *Delivery Manager's Customer Orders* data store. Thus, the customer order record is passed along the causality link between the *Delivery Manager's Log Outgoing Delivery* and the *Stock Manager's Log Books Outgoing* actions. This is denoted in Figure 6 by use of the *Note Indicator*.

Dashed lines were chosen to represent the data flow as they differentiate the data flow from the control flow, i.e. the causalities, which are represented with solid lines. Furthermore, this also avoids cluttering the action maps with solid lines.

## Organize Delivery (Initial with data stores)

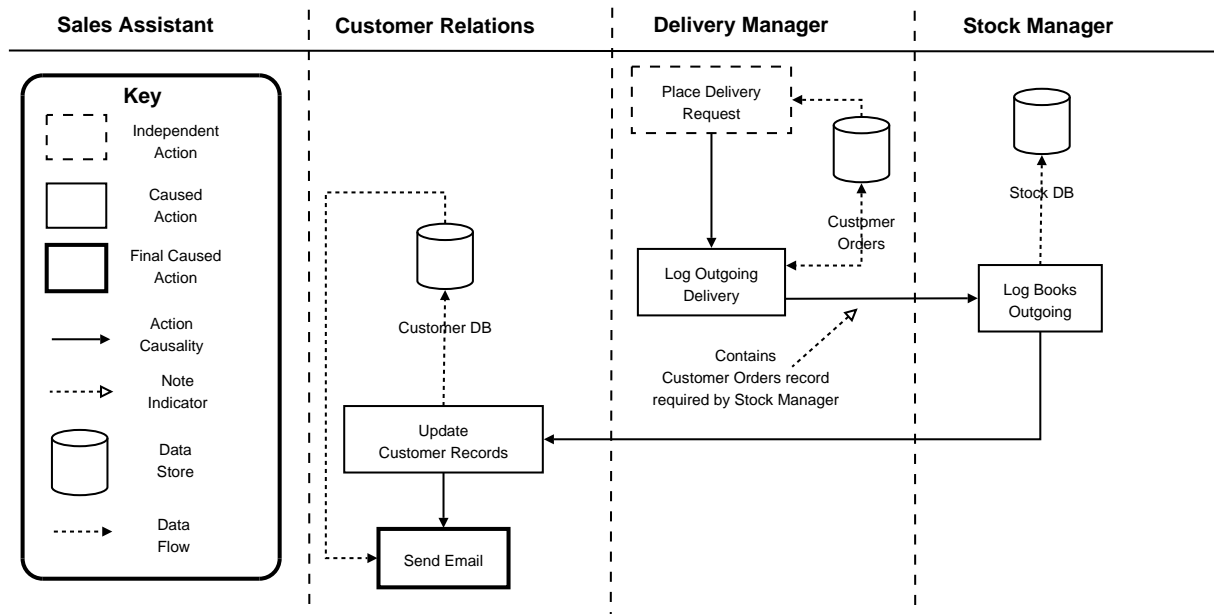


Figure 6: Action Map with Data Stores

In this step, data stores have been added and the correctness of data flow between actions has been ensured by checking that actions have access to data. The next step will generalize the action map to make it more flexible and more complete.

## Generalizing Action Maps

In this step, the designer seeks to improve the action map by generalizing it, i.e. providing multiple ways in which the action map can be followed to achieve its corresponding interaction goal. There are two ways in which this can be done. The first is for the designer to add alternative paths to success in the action map. The second is for the designer to identify where problems can occur in the action map (i.e. an action fails) and provide failure handling for the foreseen problem.

Identifying appropriate places for adding alternative paths can be difficult as it is dependent on the domain, the roles involved and the actual interaction. However, although there are no set guidelines for identifying where alternative paths can be added, the designer can systematically analyze each action and determine if the action can be achieved in a different manner or if additional useful actions can be added.

For example, in the online book store, the *Delivery Manager* does not determine if the book to be delivered is currently in stock (refer to Figure 6). Placing a delivery request can be achieved in two ways. Firstly, the availability of stock is to be checked, then, if available, the book is delivered as per the current

action map (Figure 6). However, if the book is unavailable, the book can be ordered from the publisher and once it arrives, it is then delivered to the customer.

Note that in the current action map (refer to Figure 6), the availability of the ordered book is never explicitly queried; it is assumed to be part of the delivery options. In order to clarify matters, querying for the ordered book's availability needs to be made explicit. This is done by adding two new actions at the start of the action map: *Check Book Availability* and *Check Stock* (refer to Figure 7). These two actions are used to determine how to arrange the delivery. *Check Book Availability* is used to query the *Stock Manager* about the availability of the ordered book. *Check Stock* is the action in the *Stock Manager* that replies to the query. If the ordered book is available, the delivery order is placed and processed. If the ordered book is not available, the *Add Pending Order* action is used to order the book (from the publishing firm). Once the book comes in (from the publishing firm), *Process Newly Received Stock* is triggered, the pending order is filled and the delivery is processed. The result of this step is shown in Figure 7.

### Organize Delivery (Generalized)

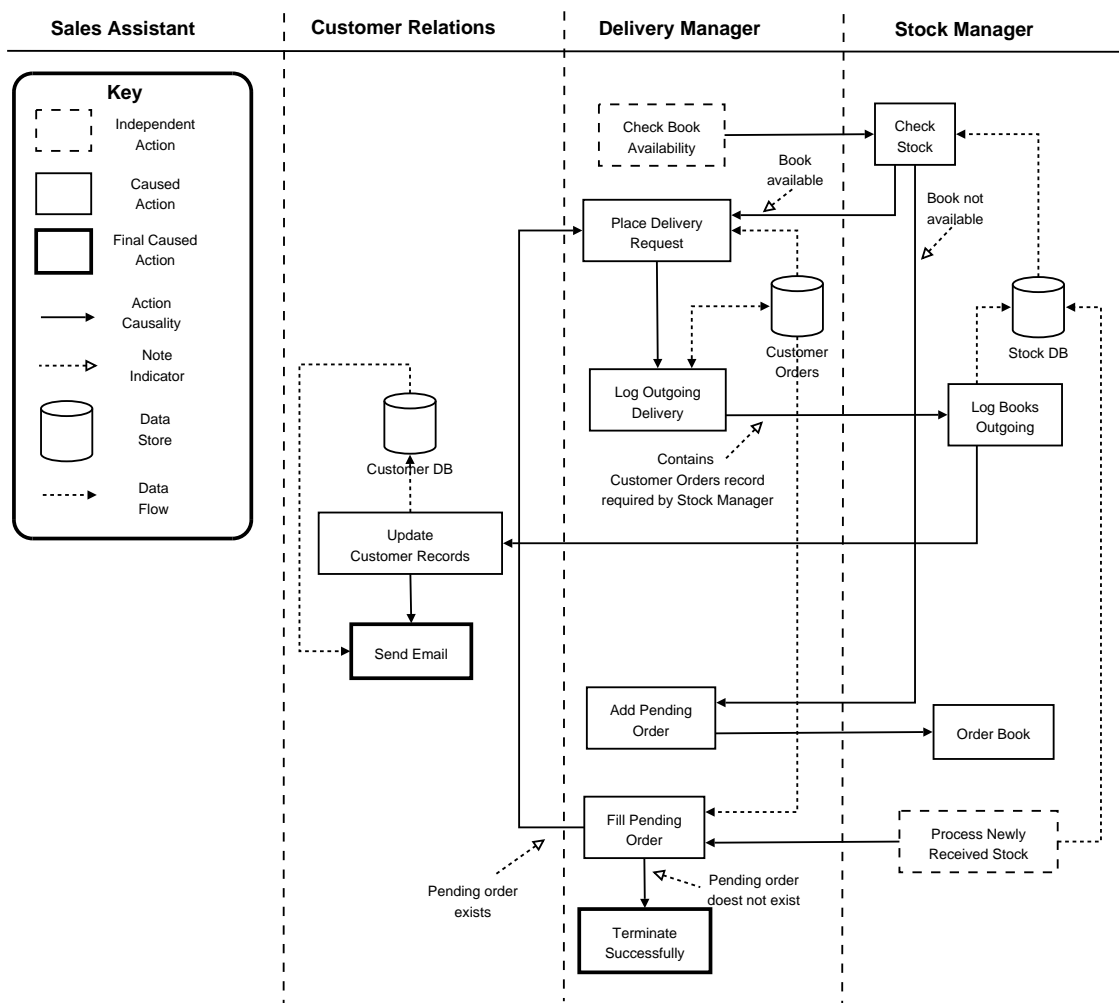


Figure 7: Generalized Action Map

The remainder of this section focuses on failures and how to attend to them. **Failure handling** is of crucial importance as it is what gives the action maps, and thus Hermes, the majority of their **robustness**<sup>11</sup>. By being able to handle foreseeable failures, the roles are able to persevere through these failures in order to complete the interaction.

The different types of failures possible in Hermes are firstly described. Then, the available Hermes failure recovery mechanisms which can be used to address these failures are presented, after which follows an explanation of how to determine and add failure handling to the action maps.

There are two types of failures in Hermes: **action failure** and **interaction goal failure**. An action failure is when an action fails to achieve its interaction goal or intended purpose. For example, *Place Delivery Request* may fail because the customer's address is invalid. An interaction goal failure is more dire. In such failures, the roles are unable to achieve the interaction goal. For example, because of incorrect credit card details, payment is unable to be processed. In this case, the *Process Bank Transaction* interaction goal fails (refer to Figure 4).

In Hermes, an **action retry** can be used to handle action failures. The concept of an action retry is simple: it allows a failed action to be recovered from by retrying that, or another, action. For example, if the *Place Delivery Request* fails because the customer's address is invalid, the customer can be asked for another address and the *Place Delivery Request* action can be retried instead of the interaction goal failing at this point.

If an action fails and is not able to be handled by action retries, this can lead to interaction goal failure. When this occurs, the interaction can be either terminated or rolled back to a previous interaction goal. The main notion of **rollback** is that if an interaction is returned to a previous interaction goal and the interaction goal is re-achieved in a different manner (which leads to a different intermediate result than previously acquired), the failed interaction goal may then be successfully re-achieved. For example, in the case of payment processing failing due to incorrect credit card details (i.e. *Process Bank Transaction* failing, refer to Figure 4), instead of terminating the interaction, the interaction can be rolled back to the *Retrieve Credit Card Details* interaction goal. That interaction goal can be re-achieved in a different manner, e.g. the customer provides the correct details of the credit card, and the *Process Bank Transaction* interaction goal will then be able to be achieved. Whereas action retry is used *within* a single interaction goal, rollback is used *between* interaction goals.

Interaction goals at which rollbacks can be issued and which interaction goals can be rolled back to is both domain- and application-specific. Therefore, it is up to the designer to determine this. The designer must thus indicate whether rollbacks are permissible for each interaction goal, and if so, which interaction goal should the interaction be allowed to roll back to.

Determining where rollbacks can be issued from can sometimes be challenging. The designer should analyze each action individually and consider whether it is sensible to issue a rollback from it if it fails. The designer should pay careful consideration to termination actions as they can often be substituted with roll backs. As a test, the designer should be able to clearly explain the purpose of issuing a rollback from

---

<sup>11</sup> Additional robustness comes from the rollback mechanism which is discussed later on.

a particular action and what advantages it brings to the interaction. Once a potential rollback has been identified, the designer will need to determine what interaction goal the rollback will roll back to.

One constraint of rollbacks is that in order to be able to roll back to a previous interaction goal, the current interaction goal must be dependent on the interaction goal which it desires to roll back to. That is, for interaction goal *B* to roll back to interaction goal *A*, *B* must depend on *A* (as *A* must occur before *B* in order for the roll back to make sense).

Similarly to rollbacks, where and when an interaction can be terminated is also domain- and application-specific. As with the rollbacks, the designer will have to carefully analyze and consider each action map and determine whether it is sensible to terminate an interaction at that point. For example it is not sensible for the online book store interaction to terminate after the payment has been taken but before the book has been delivered<sup>12</sup>.

Terminations can be identified at points in the interaction where no alternatives are possible (i.e. all possible alternatives have been exhausted) and essential particulars of the interaction cannot be agreed upon. Terminations are usually placed at points that “make or break” the interaction. If the roles involved cannot agree on a particular of the interaction and there are no alternatives, then the interaction simply cannot proceed.

Terminating in response to failure provides a graceful exit from an interaction which cannot be successfully achieved. As such, when a termination occurs, all roles involved in the interaction should leave the interaction in a desirable state. For example, in the online book store interaction, when a termination occurs, all parties should leave the interaction without incurring any loss. It would be undesirable for the customer to have made payment and for the online book store not to transfer the book.

There are three parts to adding **failure handling** to action maps:

1. Failure Identification
2. Adding Action Retries
3. Adding Rollbacks

In order to identify where possible failures might occur, the designer should think about each action and determine whether it can fail or not. If the action can fail, the designer should determine what types of failures can result from it. Once failures have been identified, the designer should determine ways in which the failures can be addressed.

For each action map, the designer should create a table to summarize all the possible failures and ways to rectify them as shown in Table 1. For example, it has been identified that the *Order Book* action could fail if a book is out of print, and the suggested remedial action is to suggest an alternative title.

---

<sup>12</sup> Unless one adds compensatory actions, in this case, to return the payment.

To further enhance **flexibility** and **robustness**, the designer can also analyse each action and determine different ways in which they can succeed (i.e. determine alternative success paths). This will further increase flexibility and robustness in the interactions.

Once failures have been identified and remedial actions determined, the designer can then update action maps with action retries and rollbacks. Adding action retries to action maps is relatively straightforward. In the case of the *Order Book* action failing, the suggested remedial action of suggesting an alternative title is achieved by the *Process Book Out Of Print Message* action, which leads to the *Suggest Alternative or Similar Title* action (refer to Figure 8). The other two identified failures and remedial actions from Table 1 have also been incorporated into Figure 8.

#	Action	Possible Failures	Remedial Actions
1	Order Book	Book out of print	Suggest alternative title or edition
2	Place Delivery Request	Invalid address	Get details from user and validate
3	Send Email	Email bounces	Use different medium to contact user (e.g. send mail via post)

*Table 1: Possible Failures and Remedial Actions for Organize Delivery*

An effect of adding **action retries** is that it can lead to loops between actions. Note that a loop has now formed between *Place Delivery* and *Get User Address*. It is important to ensure that there are no endless loops in action maps. This is done by providing an exit condition: if *Get User Address* fails (i.e. the customer cannot or does not want to provide a valid address), the interaction is terminated. Experienced designers will be able to immediately add such exit conditions but novice designers may not realize that they are required. However, novice designers should be able to identify, in a second iteration through the action maps, that the *Get User Address* action could fail. As such, this failure should be handled. In this case, it is handled by providing an action which will terminate the interaction.

Adding **rollbacks** to action maps is quite simple once they have been identified. If the desired book is unavailable and the customer wishes to purchase a suggested title, it is necessary to roll back the interaction to adjust the payment amount<sup>13</sup>. Thus, a rollback action, *Rollback to Process Bank Transaction*, is provided on the action map (refer to Figure 8) to return the interaction to the *Process Bank Transaction* interaction goal.

---

<sup>13</sup> In a typical system, payment is not normally charged before checking inventory stock, however, this system processes payment before checking inventory stock for expository purposes.

After the final iteration of this step, the action map is now in a completed state. It is also more flexible and robust than the initial action map developed in Figure 5.

## Messages

In this phase of the interaction design, messages need to be identified. These messages are necessary to realize inter-agent triggering of action/causality links as defined in action maps. To identify the messages, the designer will need to analyze the action maps and determine where one role needs to trigger an action of another role or where data needs to be transmitted from one role to another.

Consider the causality link between the *Delivery Manager's Log Outgoing Delivery* action and the *Stock Manager's Log Books Outgoing* action (refer to Figure 8). For this causality to be realized, there will need to be a message sent from the *Log Outgoing Delivery* action to the *Log Books Outgoing* action.

Part of defining messages will also involve determining the data carried by the messages. The message between the *Delivery Manager's Log Outgoing Delivery* action and the *Stock Manager's Log Books Outgoing* action will need to carry information from the customer's records. The way the data is represented will depend on the message standards being used, which, in turn, will depend on the intent of the implemented interaction. For example, if the implemented interaction is to be used in open systems, then standards such as **KQML**, **FIPA**, or **SOAP** might be appropriate. If the implemented interaction is to be used in a closed system, then the default message type of the agent platform being implemented upon might suffice.

Messages are defined in **message descriptors** which specify the message's name, type and data, as well as a description.

## Action Sequence and Action Message Diagrams

**Action sequence diagrams** and **action message diagrams** are simple and minor Hermes artifacts which can be used to check that action maps allow for desired interactions to occur. These artifacts are optional and are to be used at the designer's discretion.

An action sequence diagram follows a specific trace from an action map. It is different from action maps, which show all possible execution sequences, as an action sequence diagram shows *one* specific sequence of actions being executed. An action sequence diagram (which is similar to a UML sequence diagram) shows a lifeline for each role with the actions performed by that role placed on its lifeline. The actions are depicted using the same notation as action maps (e.g. a thick border indicates a final action). Actions that are carried out to achieve a particular interaction goal are enclosed in a shaded box which represents that interaction goal (see Figure 9).

To develop an action sequence diagram, the designer traces through the action maps and interaction goal hierarchy, and makes appropriate choices of what action is executed at particular points. That is, the designer simulates an execution of the action map.

The purpose of an action sequence diagram is to check that identified actions from the action maps are sufficient to allow for complete and successful interactions. It also allows the designer to ensure that specifically desired interactions can be generated by the interaction goal hierarchy and its associated actions.

# Organize Delivery (Final)

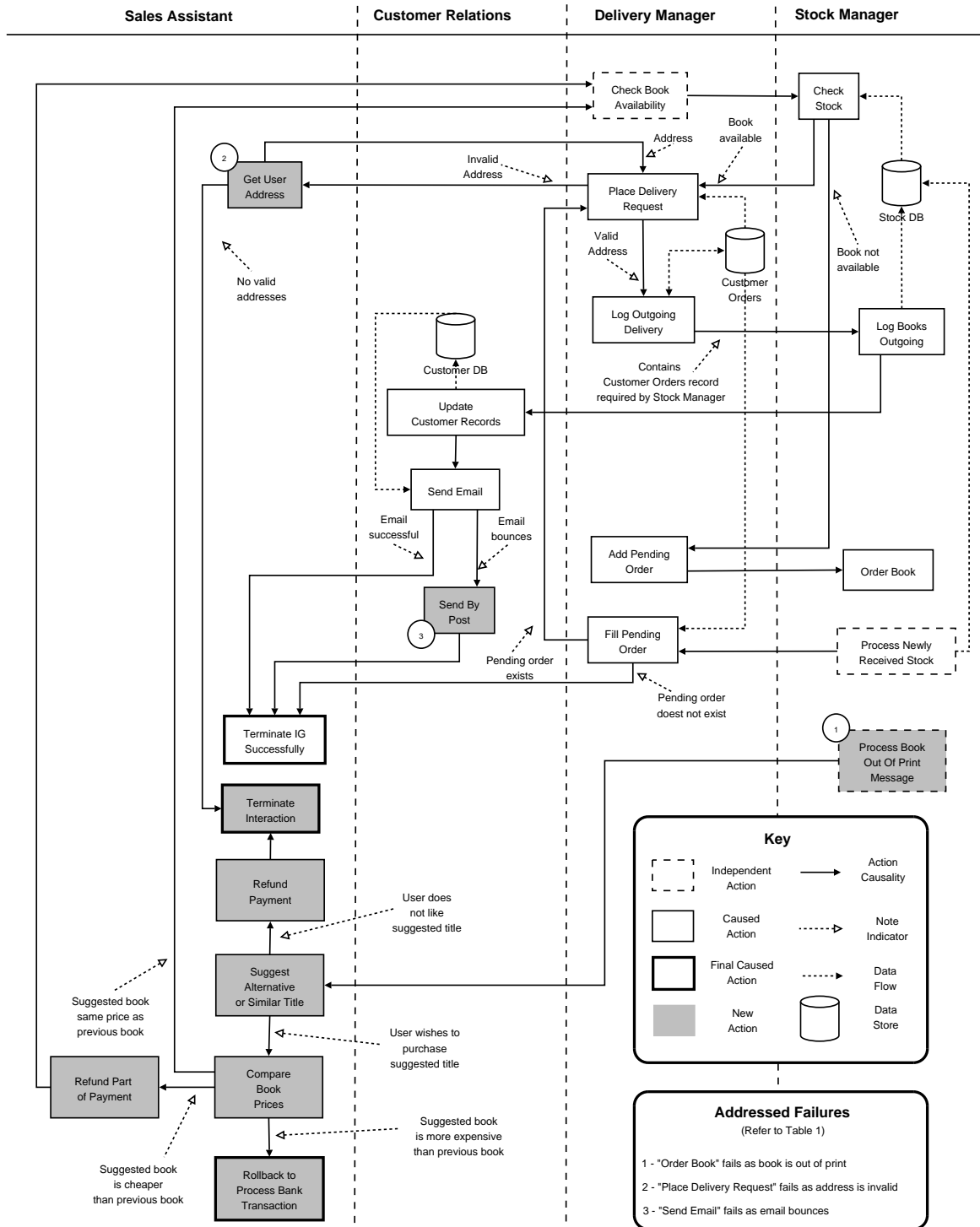


Figure 8: Finalized Action Map

Messages are added to action sequence diagrams to give action message diagrams. These can be useful in identifying what data needs to be carried in the message. When the messages are placed between the

actions, the designer can consider what data needs to be communicated between the roles. For example, in Figure 9, the request message between the *Delivery Manager's Check Book Availability* action and the *Stock Manager's Check Stock* action will need to carry across data such as the *Book-ID*.

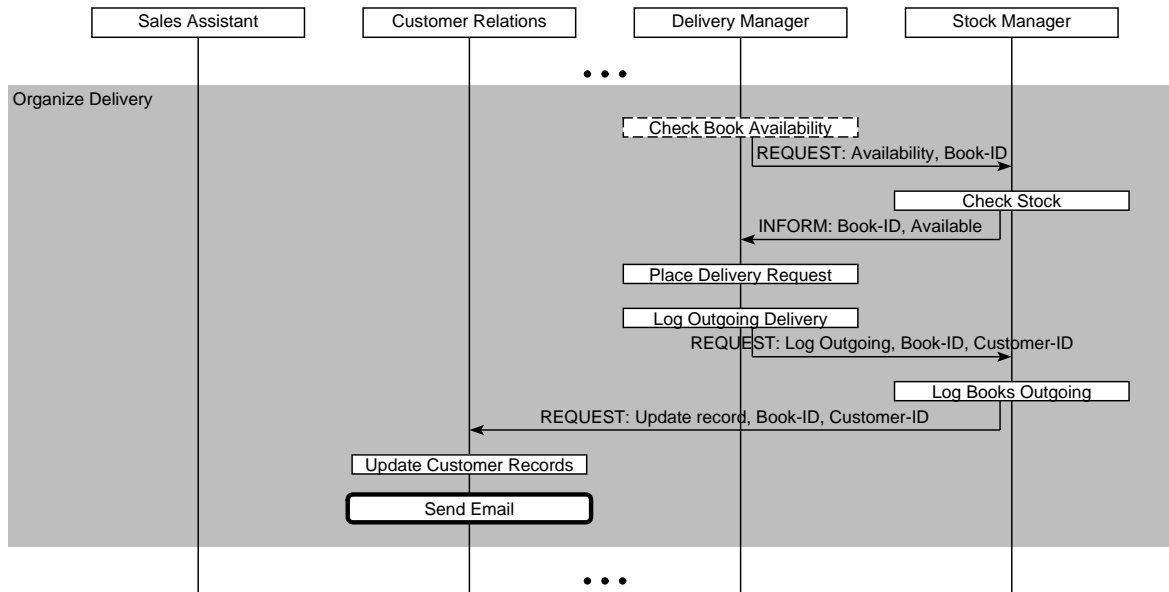


Figure 9: Action Message Diagram

## IMPLEMENTING HERMES DESIGNS

In this section, we explain how Hermes designs are implemented by mapping design artifacts to collections of goals and plans. As Hermes is goal-oriented, the implementation platform needs to be one that defines agents in terms of goals and plans. These platforms include those based on the **Belief-Desire-Intention (BDI)** model, such as **JACK**<sup>TM14</sup>, **Jadex**, **JAM**, **Jason**, and others. Although Hermes designs have only been currently implemented using Jadex, it is possible to implement Hermes on any of the aforementioned platforms. This is possible since the implementation scheme does not use any platform-specific features.

### Implementation Overview

An overview of the implementation is shown in Figure 10, including the different plan types and their inter-connections.

Interaction goals are directly mapped to coordination plans, which are used to coordinate participating agents through the interaction. These plans are common to all agents involved in the interaction. Each interaction goal in the interaction goal hierarchy is mapped to a coordination plan. The function of a coordination plan for a non-atomic interaction goal is to trigger (in an appropriate order) the plans corresponding to its child interaction goals. The function of a coordination plan corresponding to an

<sup>14</sup> JACK is a trademark of The Agent Oriented Software Group ([www.agent-software.com](http://www.agent-software.com))

atomic interaction goal is to trigger an achievement plan corresponding to an initial action in the action map for that interaction goal (thus initiating the execution of the action map).

Achievement plans are derived from actions in the action maps and differ from agent to agent. They provide steps which the agents take towards achieving an interaction goal. The collection of achievement plans corresponding to an action map together realise the interaction described by the action map.

At runtime each agent has a hierarchy of coordination plans corresponding to the current state of the interaction. For example, considering the interaction goal hierarchy in figure 4, the interaction begins by instantiating a coordination plan corresponding to the root interaction goal (*Order Book*). This interaction goal triggers a coordination plan corresponding to the interaction goal *Retrieve Details*. Once this is complete (which involves more coordination plans and achievement plans), the coordination plan for *Order Book* triggers the coordination plan for *Process Order* which in turn results in coordination plans being triggered for *Process Bank Transaction* and *Organize Delivery*. The coordination plan for *Organize Delivery* will trigger the initial action of the relevant action map (*Check Book Availability*).

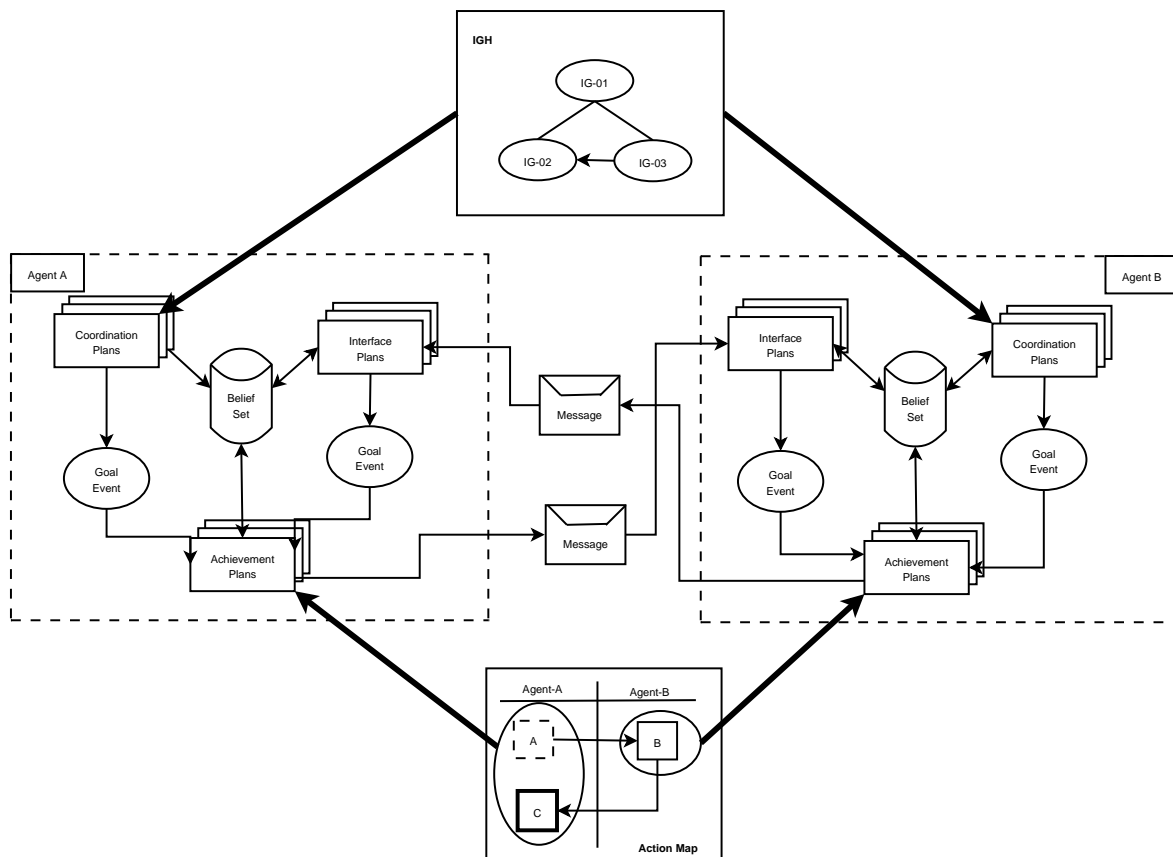


Figure 10: Implementation Overview

The third type of plans are Interface plans, which are derived from the message descriptors and action maps. They are used to transform inter-agent messages into goals and events for intra-agent processing. Each incoming message is mapped to an internal event that triggers the appropriate plans. For example, when the *Delivery Manager* sends a request to the *Stock Manager* to check the availability of a book (refer to Figure 8), the *Stock Manager's* interface plan converts the message into the internal event that will trigger the plan that corresponds to the appropriate action in the action map (*Check Stock*). The interface plans also check whether the agent has initialised the interaction, and if not, they trigger the creation of a hierarchy of coordination plans corresponding to the interaction goal hierarchy.

The following sections explain the representation of interaction goals, and the different plan types.

### Interaction Goal Representation and Beliefs

As can be seen in Figure 10, agent beliefs connect the different plan types. Beliefs are used to pass information between plans so they can coordinate the agents through the interaction. The states of interactions goals are represented using a combination of three Boolean values per interaction goal: *in*, *finished*, and *success*. The *in* belief indicates that the interaction goal is currently active. The *finished* belief is used to indicate whether the interaction goal has been completed, whilst *success* (or *succeeded*) indicates whether the interaction goal has been successful.

The interaction goal states and valid transitions between them are shown in Figure 11. The dashed circles represent intermediate states that have no conceptual meaning, but are necessary to change state from *active* to either *succeeded* or *failed*. The Boolean string in parentheses show the values of the three beliefs *in*, *finished*, and *success*, respectively. Transitions between these states are triggered by the coordination and achievement plans (see algorithms 1-4 below).

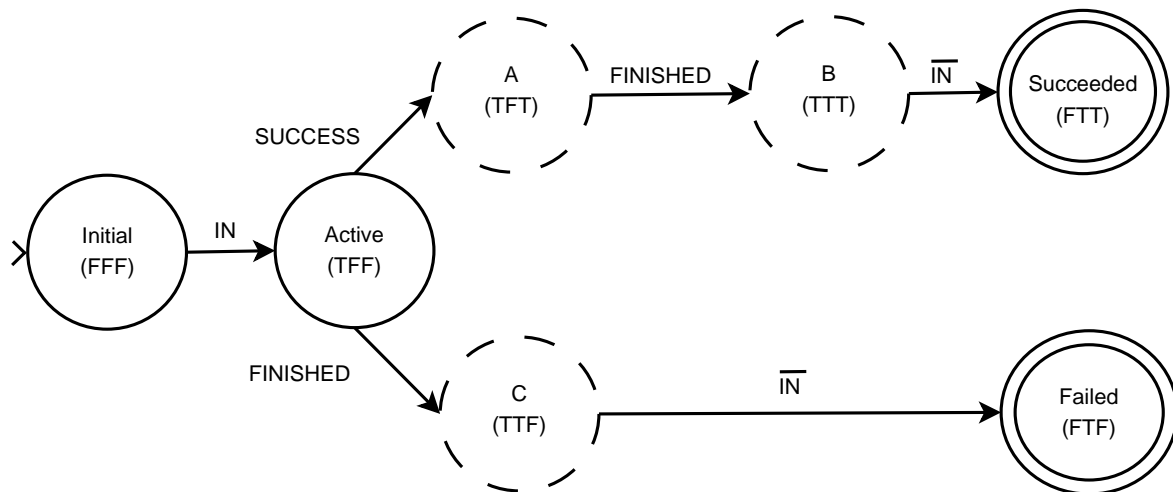


Figure 11: Interaction Goal States and Valid Transitions

Beliefs are also used to represent a number of other attributes in interactions. A summary of the agent's beliefs, along with examples, is shown in Table 2. The examples are presented as `key:value` pairs with default values for the given interaction selected.

Belief	Use	Example
role	Identifies the agent's role in the interaction.	role:deliveryManager
initiator	Identifies the interaction's initiator.	initiator:deliveryManager
Interaction Goal Initiators	A series of beliefs which identify the initiator of each IG (one per IG).	orderBookIGInitiator:salesAssistant processOrderIGInitiator:deliveryManager
Interaction Goal States	A series of beliefs used to represent the state of IGs, i.e. in, finished, and success. Used for Coordination-Achievement plan connections.	inOrderBookIG:false finishedOrderBookIG:false OrderBookIGSuccess:false
Interaction Goal Retries	A series of beliefs for retrying IGs. One for each IG that is allowed to be retried.	retryProcessBankTransaction:false
Interaction Specific Beliefs	Beliefs which are specific to the given interaction.	bookID:20 customerID:7

Table 2: Belief Structure and Examples

For each interaction, each agent has a *role* belief which states the name of the agent's role in the interaction (e.g. the *Delivery Manager* role in the online book store interaction). The *role* beliefs are generally used to determine which roles need to take action in each interaction goal. The *initiator* role states which role initiated the interaction. This is needed for interaction goals in which the initiator is inherited.

The interaction goal initiator beliefs are a series of beliefs which identify an initiator for each interaction goal. The initiator role is responsible for taking the initial action for its designated interaction role, which will cause the other agents involved to take responsive actions and achieve the interaction goal.

The interaction goal retries are used to flag that their respective interaction goals are being retried. This is important as the agents will then try to achieve a different outcome than that achieved previously so that the interaction can proceed successfully. For more details on the action retry failure handling mechanism, see the section on achievement plans below. The remainder of the beliefs are interaction specific, including beliefs that are based on data stores from the action maps.

## Coordination Plans

Coordination plans, derived from interaction goals, are a common set of plans that guide the agents through their interactions. There are two variations of coordination plans, *compound* and *atomic*. Compound coordination plans are based on compound interaction goals, i.e. those that are composed of other interaction goals, such as *Order Book*, *Retrieve Details*, and *Process Order* (refer to Figure 4), whilst atomic coordination plans are derived from interaction goals that are not composed of other interaction goals (i.e. atomic interaction goal), such as *Retrieve Delivery Choice*, *Retrieve Credit Card Details*, *Process Bank Transaction*, and *Organize Delivery*. Compound coordination plans are involved with coordination between themselves and other coordination plans whilst atomic coordination plans deal with coordination between themselves and *achievement* plans.

In the implementation of a Hermes interaction, there is a coordination plan for each interaction goal. Algorithm 1 presents an example of a generic compound coordination plan. As this is a generic coordination plan, it is assumed that it is derived from an interaction goal named IG.

The following beliefs are initialized to `false` for each interaction goal IG when the agent is created: `in[IG]`, `finished[IG]` and `succeeded[IG]`. The trigger to execute a compound coordination plan is its `in` belief. In this case, it is when the `in` belief changes to `true` (as shown by the `Require` statement in Algorithm 1).

When a compound coordination plan is executed, its first step is to begin the achievement of its sub-coordination plans (i.e. child interaction goals) in the specified order<sup>15</sup>. Algorithm 1 shows an interaction in which all child interaction goals are to be achieved in sequence (denoted by the *while* loop). As such, the IG coordination plan begins by retrieving the name of the next child interaction goal to be achieved (line 5) The coordination plan then sets the `in` belief of the child interaction goal to `true` (line 8), which allows the child interaction goal's coordination plan to execute. The IG coordination plan then waits until the child interaction goal is achieved (line 9), and then attempts to achieve the following child interaction goal if the current one has been achieved successfully (lines 10-12).

### Algorithm 1: Generic (Sequential) Compound Coordination Plan for Interaction Goal IG

```
Require: in[IG] ==true
1. terminate = false
2.
3. while moreChildIGs() and not terminate
4.     // Get beliefs for next IG
5.     ChildIG = nextChildName()
6.
7.     // Coordination
8.     in[ChildIG] = true
9.     waitFor(finished[ChildIG] and not in[ChildIG])
10. if not succeeded[childIG] then
```

<sup>15</sup> In this example we assume a sequential order. Space precludes a detailed discussion of parallelism, but see (Cheong, 2008) for details.

```

11.     terminate = true
12.   end if
13. end while
14.
15. if all child IGs succeeded then
16.   succeeded[IG] = true
17. end if
18.
19. // Synchronization (with other Coordination plans)
20. finished[IG] = true
21. in[IG] = false

```

When all its child interaction goals have been successfully achieved, the IG coordination plan sets its succeeded belief to true (lines 15-17). The last part of the compound coordination plan is to synchronize itself with the other coordination plans. That is, it sets its in and finished beliefs (in this case in[IG] and finished[IG]) to false and true respectively to signal its completion.

Algorithm 2 is an example of a generic atomic coordination plan. As with compound coordination plans, atomic coordination plans are triggered when their in beliefs change to true (refer to Require statement in Algorithm 2).

**Algorithm 2:** Generic Atomic Coordination Plan for Interaction Goal IG

```

Require: in[IG] == true
1. if not succeeded[IG] then
2.   if role==initiator then
3.     dispatch(new triggerInitialActionGoal())
4.   end if
5. end if
6.
7. // Synchronisation (with Achievement plans)
8. waitFor(finished[IG])
9. in[IG] = false

```

The first step of an atomic coordination plan is to execute the initial action (in the relevant action map) in an attempt to achieve itself. However, before that action is triggered, the atomic coordination plan ensures that it has not already been achieved (refer to line 1). This is important in situations where rollbacks are issued – there is no need to achieve a coordination plan that is already achieved. Furthermore, the atomic coordination plan ensures that only the initiator of this interaction goal begins the interaction (refer to line 2). If these conditions are met, the coordination plan triggers the initial action by dispatching the appropriate goal (line 3).

Once the goal has been dispatched, the atomic coordination plan must wait until the action (which is likely to trigger a series of other actions) completes. When the series of actions is completed, the finished belief of the current interaction goal will be set to true. Thus, as part of its synchronization

with the achievement plans (implementations of the actions), the atomic coordination plan waits for its finished belief to be set to `true` (line 8). The last part of the atomic coordination plan is to set its `in` belief to `false`, signifying that it has been completed.

## Achievement Plans

Achievement plans, based on actions from the action maps, are used by the interacting agents as steps towards achieving an interaction goal. Therefore, achievement plans usually contain interaction-specific steps.

Algorithm 3 presents an example of a generic achievement plan. For an achievement plan to begin execution, it must be triggered by an appropriate goal event, as shown by the `Require` statement in Algorithm 3. Once triggered, the achievement plan must ensure that it is in the correct context, i.e. its interaction goal is active, before beginning to execute (refer to line 2). When the achievement plan is in the correct context, it executes interaction-specific code (lines 4 and 5).

### Algorithm 3: Generic Achievement Plan

**Require:** `actionTriggerGoalEvent`

```
1. // Synchronisation (with Coordination plan)
2. waitFor(in[IG])
3.
4. // AchieveIG (application specific)
5. ...
6. if action achieved IG then
7.   succeeded[IG] = true // Action achieves IG
8. end if
9.
10. // Finish IG, only done if action is final
11. // Synchronisation (with Coordination plan)
12. if action is final then
13.   finished[IG] = true
14. end if
```

If the achievement plan represents an action that achieves the interaction goal (i.e. a final action that terminates with success), then the interaction goal's `success` belief is set to `true` (lines 6-8).

Furthermore, achievement plans representing final actions have a synchronization section which sets the `finished` belief of the interaction goal to `true`, signaling the completion of the interaction goal (lines 10-14).

The implementation of action failure handling mechanisms, *termination* and *action retry*, are simple and straightforward. When an action fails (i.e. an achievement plan fails), there are two options: terminate the interaction or attempt to recover by retrying the action with different parameters.

This first option, termination, is the simplest. In such a case, the programmer will need to add actions to request termination and to terminate the interaction.

For example, in Figure 8, the *Sales Assistant* will terminate the interaction if the customer is unable to provide a valid address or does not wish to purchase a suggested alternative book. To implement the termination, each role in the interaction is equipped with an action to request termination, i.e. *Request Termination*, and an action to actually terminate the interaction, i.e. *Terminate Interaction*. The termination is a chain-like sequence as follows. The *Sales Assistant* requests termination from the *Customer Relations*, which then requests termination from the *Delivery Manager*, which then requests termination from the *Stock Manager*. The *Stock Manager* then terminates its interaction and notifies the *Delivery Manager*, which in turn terminates its interaction and notifies the *Customer Relations*, which also terminates its interaction and notifies the *Sales Assistant*, which then terminates its interaction<sup>16</sup>.

This sequence of actions can be quite easily added to the action map, however, to avoid unnecessarily cluttering the diagram, by convention, the *Terminate Interaction* action (in Figure 8) is understood to represent this sequence of actions. This is similar to the sequence used to specify rollback, as depicted in Figure 12.

Implementing the **rollback** failure handling mechanism, which addresses interaction goal failure, is more complicated than implementing terminations or action retries. Algorithm 4 is an example of a rollback achievement plan in which the *Sales Assistant* is rolling back from the *Organize Delivery* interaction goal to the *Process Bank Transaction* interaction goal as per the action map in Figure 8. The comments (in bold) present a general plan for rolling back with the code showing how the *Sales Assistant* rolls back in this particular example.

In general, a rollback is implemented by “saving” the interaction in a particular state and re-starting the entire interaction. The “saving” of the interaction is done by setting the interaction goal to which the agent wishes to roll back to be attempted next. This is done by setting its `in` belief to `true` and both its `finished` and `success` beliefs to `false`, which essentially flags the interaction goal as active, but not yet completed (lines 17-20 in Algorithm 4). Thus, it will be attempted next (unless there are other active but uncompleted interaction goals preceding it).

When the interaction is re-started, the agents will not re-attempt interaction goals that have already been successfully achieved. Thus, the agent will re-attempt the desired interaction goal next.

In Algorithm 4, the agent must firstly ensure that it is in the correct context (i.e. its current interaction goal is active) before it can carry out the rollback. In this case, the *Sales Assistant* must wait until the *Organize Delivery* interaction goal is active (lines 1 and 2). The agent then terminates the current interaction goal by setting its `in` and `success` beliefs to `false` and its `finished` belief to `true`. This will cause the interaction to fail, which the agent waits for (lines 9 and 10). After the interaction has failed, the agent sets the appropriate beliefs to re-start the interaction at the desired interaction goal (this is rollback-specific) and then re-starts the interaction.

---

<sup>16</sup> In this example, the interaction is terminated sequentially. It is also possible to terminate the interaction in parallel.

**Algorithm 4:** Sales Assistant Rollback Plan (from *Organize Delivery* to *Process Bank Transaction*)

```
Require: rollbackGoalEvent
1. // Synchronise (with Coordination plan)
2. waitFor(inOrganizeDelivery)
3.
4. // 1. Terminate current IG unsuccessfully
5. organizeDeliverySuccess = false
6. finishedOrganizeDelivery = true
7. inOrganizeDelivery = false
8.
9. // 2. Wait for apex IG to terminate
10. waitFor(finishedOrderBook and not inOrderBook)
11.
12. // 3. Set appropriate beliefs to re-start interaction to begin at desired IG (shortcut)
13. // 3.1. Reset current IG beliefs
14. finishedOrganizeDelivery = false
15.
16. // 3.2. Set beliefs of IG to begin next interaction from (shortcut)
17. processBankTransactionSuccess = false
18. finishedProcessBankTransaction = false
19. inProcessBankTransaction = true
20.
21. // 3.3. Set beliefs for "retry" attempt
22. retryProcessBankTransaction = true
23.
24. // 4. Re-start interaction, set "in" belief of apex stage to "true"
25. inOrderBook = true
26.
27. // 5. Notify relevant agents
```

In Algorithm 4 the *Sales Assistant* does not need to notify any agents (line 27) as it is the last agent to roll back (refer to Figure 12). However, in the case of all the remaining agents in the interaction in Figure 12, they would have to notify another agent that they have completed their rollback so that the next agent can then begin its rollback.

As with terminations, a similar sequence of actions is, by convention, understood by the Rollback to *Process Bank Transaction* action in Figure 8. That is, the *Sales Assistant* will request that the *Customer Relations* roll back to *Process Bank Transaction*, the *Customer Relations* will send a similar request to the next agent in the interaction, and so forth, until the last agent receives the request and rolls back. After rolling back, the last agent will notify the previous agent, which will then roll back and notify the

previous agent, and so forth, until the *Sales Assistant* receives notification that all agents have rolled back. The *Sales Assistant* will then roll back. Refer to Figure 12 for a diagrammatical depiction of this sequence of actions.

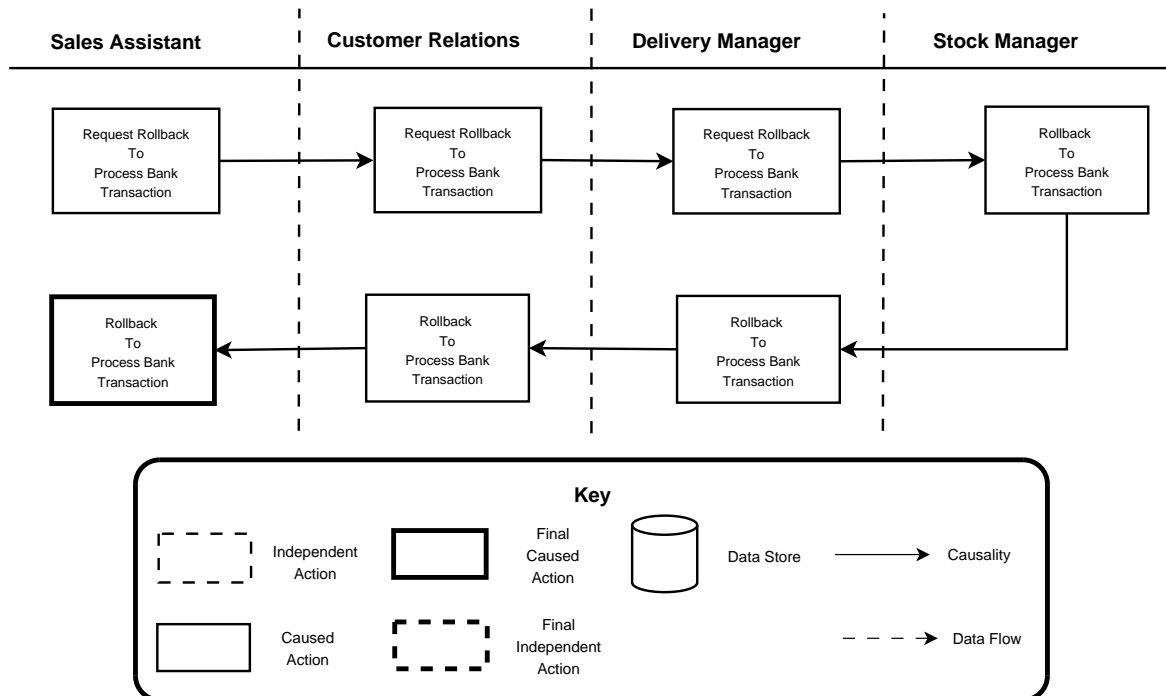


Figure 12: Rollback Sequence

## CONCLUSIONS

We have presented Hermes, a goal-oriented methodology for designing and implementing flexible and robust intelligent agent interactions. The methodology includes design processes, design notations, design artifacts, and implementation guidelines which explain how the design artifacts can be mapped to any goal-plan agent platform.

As Hermes aims to be pragmatic, this has been assessed in an **experimental evaluation** (see (Cheong, 2008) for full details) in which 13 participants were given a common interaction scenario to design. The participants were firstly given a pre-evaluation questionnaire which assessed their skill and experience with agent interaction design. Based on the responses, the participants were equally, both in terms of numbers, and skill and experience level, divided into two groups. Both groups were given the same interaction scenario to design, however, one group was asked to use the **Hermes** methodology, whilst the other used the interaction design aspect of the **Prometheus** methodology. Each participant was provided with the appropriate training manual (Hermes or Prometheus). After completing their design interactions, participants were required to fill in a post-evaluation questionnaire which enquired about how they felt about their created design interactions, and how they felt about the processes they used to create them. In addition, the designs themselves were collected and analysed.

The participants' designed interactions were analysed with respect to four metrics: *Scenario Coverage*, *Flexibility*, *Robustness*, and *Design Time*. The first metric (*scenario coverage*) was used to assess how

well the methodology guided the designer. Specifically, we considered whether any of the steps in the provided scenario were missed in the interaction, something that a good process should help the designer avoid. The next two metrics (*flexibility* and *robustness*) directly measured how successful the methodologies were at guiding developers towards producing flexible and robust interactions. The flexibility metric assessed the number of possible paths through the interaction (as a function of domain-specific interaction parameters, such as the number of possible meeting times). The robustness metric assessed the number of possible failures that were considered in the design. Finally, the design time metric simply measured how long it took to produce the design.

The results of the evaluation indicated that Hermes was successful:

- **Scenario coverage:** all of the designs produced with Hermes covered all 14 steps of the scenario whereas more than half of the designs produced by following Prometheus did not cover between 1 and 4 steps (difference being statistically significant<sup>17</sup> with  $p=0.04895$ )
- **Flexibility:** from each design we derived the number of paths through the interaction as a function of the number of alternative meeting times ( $m$ ), the number of alternative rooms ( $r$ ), and the number of alternative credit cards ( $c$ ). Considering a range of reasonable values for these demonstrated significant differences (e.g.  $p=0.01632$  for  $m=c=r=3$ ) with the number of paths ranging from 4 to 2655 for Prometheus designs and from 164 to 405872 for Hermes (both with  $m=c=r=3$ ).
- **Robustness:** we identified nine failures that could occur in the course of the interaction (but did not provide these to the designers). Of these nine, Prometheus designs identified 0-3 whereas Hermes designs identified 3-7, demonstrating better robustness in Hermes designs ( $p=0.001748$ )

However, one disadvantage of Hermes was that it took longer to follow the methodology (Prometheus designs ranged from 45 to 240 minutes, Hermes from 145 to 320,  $p=0.006993$ ). An interesting observation, that was substantiated by the participants in their responses to the post-evaluation questionnaire, was that the way Hermes divides interactions, per interaction goals, was easier to follow than Prometheus' per agent division of the interaction. Otherwise, results from the post-evaluation questionnaire supported the evaluation results but were less conclusive, since they considered participants' opinions about the designs, as opposed to considering the designs themselves, and since they had a limited seven point (+3 to -3) response scale.

## FUTURE RESEARCH

There are two tools to support the Hermes methodology. One is a simple design tool that allows designers to create interaction goal hierarchies and action maps, whilst the other is a code generation tool which accepts a syntactic description of a Hermes interaction and produces (partial) **Jadex** code. One area of future work is to improve tool support for Hermes, including:

- improvements in support for the processes and techniques, rather than just providing a 'sketch pad' for the Hermes notations;

---

<sup>17</sup> Statistical significance was tested using an exact Wilcoxon rank sum test.

- adding checking of designs for consistency; and
- integrating Hermes support into the Prometheus Design Tool (PDT).

As Hermes is purposely limited to the design of agent interactions, it has been integrated with Prometheus to enable designers to use goal-oriented interactions in agent system design (Cheong and Winikoff, 2006b). Apart from Prometheus, Hermes can also be integrated with other agent methodologies, however, this has not been done, and remains an area for future work.

Although Hermes was not explicitly created to design open systems interactions, this has been kept in mind during its development. As such, it should be possible to adapt Hermes to design interactions in open systems. This would require changing the implementation mapping to cater for non-BDI agents, but should not require significant changes to the design methodology. Additionally, Hermes can also be used to design interactions between and within teams, however, this work has not yet been researched.

Finally, there is scope for developing better techniques and notations for dealing with parallelism in interaction design.

## REFERENCES

- Bauer, B., Müller, J. P., and Odell, J. (2000). An extension of UML by protocols for multi-agent interaction. In *Proceedings of the Fourth International Conference on MultiAgent Systems*, pages 207-214.
- Bauer, B., Müller, J. P., and Odell, J. (2001). Agent UML: A formalism for specifying multiagent software systems. In *First International Workshop on Agent-Oriented Software Engineering (AOSE 2000)*, pages 91-103, Secaucus, NJ, USA. Springer-Verlag New York, Inc.
- Benfield, S. S., Hendrickson, J., and Galanti, D. (2006). Making a strong business case for multiagent technology. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*, pages 10-15, Hakodate, Japan. ACM Press.
- Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., and Perini, A. (2004). Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, 8(3):203-236.
- Castelfranchi, C. (1995). Commitments: From individual intentions to groups and organizations. In Lesser, V. R. and Gasser, L., editors, *Proceedings of the First International Conference on Multiagent Systems*, pages 41-48, San Francisco, California, USA. The MIT Press.
- Chaib-draa, B. (1995). Industrial applications of distributed AI. *Communications of the ACM*, 38(11):49-53.
- Cheong, C. (2008). Hermes: Goal-oriented interactions for intelligent agents. PhD thesis. School of Computer Science and Information Technology, RMIT University. (In preparation).

- Cheong, C. and Winikoff, M. (2006a). Hermes: Designing goal-oriented agent interactions. In Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Programming Multi-Agent Systems (ProMAS 2005 post-proceedings)*, volume 3862 of *Lecture Notes in Artificial Intelligence*, pages 168-183. Springer.
- Cheong, C. and Winikoff, M. (2006b). Improving flexibility and robustness in agent interactions: Extending Prometheus with Hermes. In *Software Engineering for Multi-Agent Systems IV: Research Issues and Practical Applications (SELMAS 2005 post-proceedings)*, volume 3914 of *Lecture Notes in Computer Science*, pages 189-206. Springer.
- DeLoach, S. A. (2006). Engineering organization-based multiagent systems. In Garcia, A. F., Choren, R., de Lucena, C. J. P., Giorgini, P., Holvoet, T., and Romanovsky, A. B., editors, *Software Engineering for Multi-Agent Systems IV: Research Issues and Practical Applications (SELMAS 2005 post-proceedings)*, volume 3914 of *Lecture Notes in Computer Science*, pages 109-125. Springer.
- DeLoach, S. A., Wood, M. F., and Sparkman, C. H. (2001). Multiagent systems engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 11(3):231-258.
- Dignum, V. (2004). *A model for organizational interaction: Based on agents, founded in logic*. PhD thesis, Utrecht University, The Netherlands.
- Dignum, V. and Dignum, F. (2003). The knowledge market: Agent mediated knowledge sharing. In *Proceedings of the Third International/Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 03)*, pages 168-179.
- Esteva, M., de la Cruz, D., and Sierra, C. (2002). Islander: An electronic institutions editor. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '02)*, pages 1045-1052, Bologna, Italy. ACM Press.
- Flores, R. A. and Kremer, R. C. (2004a). A pragmatic approach to build conversation protocols using social commitments. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, pages 1242-1243, New York, NY, USA. ACM Press.
- Flores, R. A. and Kremer, R. C. (2004b). A principled modular approach to construct flexible conversation protocols. In Tawfik, A. and Goodwin, S., editors, *Advances in Artificial Intelligence*, pages 1-15. Springer-Verlag, LNCS 3060.
- Fornara, N. and Colombetti, M. (2002). Operational specification of a commitment-based agent communication language. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '02)*, pages 536-542, Bologna, Italy. ACM Press.
- Fornara, N. and Colombetti, M. (2003). Defining interaction protocols using a commitment-based agent communication language. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '03)*, pages 520-527, Melbourne, Australia. ACM Press.
- Huget, M.-P. and Odell, J. (2004). Representing agent interaction protocols with agent UML. In *Proceedings of the Fifth International Workshop on Agent Oriented Software Engineering (AOSE)*, pages 16-30, Springer-Verlag, LNCS 3382.

- Huget, M.-P., Odell, J., Haugen, Ø., Nodine, M. M., Cranefield, S., Levy, R., and Padgham, L. (2003). FIPA Modeling: Interaction Diagrams. On [www.auml.org](http://www.auml.org) under “Working Documents”. Foundation For Intelligent Physical Agents (FIPA) Working Draft (version 2003-07-02).
- Hutchison, J. and Winikoff, M. (2002). Flexibility and Robustness in Agent Interaction Protocols. In *Workshop on Challenges in Open Agent Systems* at the First International Joint Conference on Autonomous Agents and Multi-Agents Systems.
- Jennings, N. R. (2001). An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35-41.
- Jennings, N. R., Sycara, K. P., and Wooldridge, M. (1998). A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7-38.
- Kumar, S. and Cohen, P. R. (2004). STAPLE: An agent programming language based on the joint intention theory. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS '04)*, pages 1390-1391, New York, USA. ACM Press.
- Kumar, S., Cohen, P. R., and Huber, M. J. (2002a). Direct execution of team specifications in STAPLE. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS '02)*, pages 567-568, Bologna, Italy. ACM Press.
- Kumar, S., Huber, M. J., and Cohen, P. R. (2002b). Representing and executing protocols as joint actions. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS '02)*, pages 543-550, Bologna, Italy. ACM Press.
- Munroe, S., Miller, T., Belecheanu, R. A., Pěchouček, M., McBurney, P., and Luck, M. (2006). Crossing the agent technology chasm: Lessons, experiences and challenges in commercial applications of agents. *Knowledge Engineering Review*, 21(4):345-392.
- Odell, J., Parunak, H. V. D., and Bauer, B. (2000). Extending UML for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop* at the 17th National conference on Artificial Intelligence, pages 3-17, Austin, TX.
- Padgham, L. and Winikoff, M. (2004). *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons. ISBN 0-470-86120-7.
- Reisig, W. (1985). *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag. ISBN 0-387-13723-8.
- Singh, M. P. (1991). Social and psychological commitments in multiagent systems. In *AAAI Fall Symposium on Knowledge and Action at Social and Organizational Levels*, Monterey, California, USA.
- Sycara, K. P. (1998). Multiagent systems. *AI Magazine*, 19(2):79-92.
- Vasconcelos, W., Sierra, C., and Esteva, M. (2002). An approach to rapid prototyping of large multi-agent systems. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 13-22.

Winikoff, M. (2006). Designing commitment-based agent interactions. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-06)*, pages 363-370, Hong Kong.

Winikoff, M. (2007). Implementing commitment-based interactions. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007)*, pages 868-875, Honolulu, USA.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. John Wiley and Sons.

Yolum, P. (2005). Towards design tools for protocol development. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '05)*, pages 99-105, Utrecht, The Netherlands. ACM Press.

Yolum, P. and Singh, M. P. (2001). Commitment machines. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL 2001)*, Seattle, WA, USA. Pages 235-247 in post-proceedings published by Springer-Verlag (2002), LNCS 2333.

Yolum, P. and Singh, M. P. (2002). Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proceedings of the 1st Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS '02)*, pages 527-534, Bologna, Italy. ACM Press.

Yolum, P. and Singh, M. P. (2004). Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and Artificial Intelligence (AMAI), Special Issue on Computational Logic in Multi-Agent Systems*, 42(1-3):227-253.

F. Zambonelli, N. R. Jennings, and M. Wooldridge (2003). Developing Multiagent Systems: The Gaia Methodology. *ACM Transactions on Software Engineering Methodology*, 12(3):317-370.

## KEY TERMS

**Flexibility:** An agent is said to exhibit *flexible* behaviour if it can achieve its goals in a range of ways, depending on the situation at hand.

**Interaction Goal:** An interaction goal is a goal that is achieved by two or more agents interacting. It can be seen as the goal of the interaction.

**Message-centric design:** An approach for conceptualising and designing interactions that focuses on the messages exchanges, as opposed to, for example, focussing on the social commitments that drive the exchange of messages.

**Robustness:** An agent is said to be *robust* if it is able to recover from failures. A flexible agent can exploit its flexibility to recover from failure by trying alternative means to realise its goals should failure occur.

**Social Commitment:** A promise (obligation) by one agent to another agent. There are two “flavours” of social commitments: commitments to perform action, and commitments to bring about certain conditions.

**Software Engineering Methodology:** An approach (collection of practices) for developing software. Typically covers analysis and design activities. Can include notations and choice of design models, processes to be followed, and “techniques” and heuristics for carrying out steps of the process.