AgentC: A Compiled Agent Programming Language

Henrik Lauritzen

LYNGBY 2002
EKSAMENSPROJEKT
NR. 61/02

IMM

Abstract

For more than a decade agents and (multi-)agent systems have been subject to extensive research. However, this research has mostly been focused on theoretical areas, and has to some degree neglected the issue of making agent programs usable (and useful) in practice.

This thesis introduces the AgentC Toolkit (ACT), a software toolkit designed to aid the construction of agent software, and to encourage experiments with agent based software systems.

The core of the ACT is AgentC, an agent programming language which allows direct compilation into Java™ source code; the resulting code can then easily be integrated into a user application.

Some fundamental properties of AgentC are borrowed from earlier agent programming languages like Agent-0 and PLACA. In contrast to such languages, however, AgentC is not founded on a highly specialised formal logic, but does in many respects bear a closer resemblance to traditional procedural programming languages.

Keywords: Intelligent agents, Programming languages, Compilers, Agent programs, Object-oriented programming.

Sammendrag

Agenter og (multi-)agentsystemer har været genstand for forskning igennem mere end ti år. Forskningen har dog for en stor dels vedkommende været rettet mod områder af hovedsageligt teoretisk interesse, hvorfor det i nogen grad er blevet forsømt at adressere de problemer der er forbundet ved at realisere og anvende agentprogrammer i praksis.

Dette eksamensprojekt omhandler et programbibliotek kaldet ACT (en forkortelse af AgentC *Toolkit*); programbibliotekets formål er at gøre det lettere at konstruere agentprogrammer, samt at anspore til eksperimenter med agentbaserede programsystemer.

Hovedbestanddelen i ACT er AgentC, et agentprogrammeringssprog som tillader direkte oversættelse til Java™-programkode; denne programkode kan efterfølgende let integreres i en brugerdefineret applikation.

Visse fundamentale træk ved AgentC er i høj grad inspireret af tidligere agentprogrammeringssprog såsom Agent-0 og PLACA. I modsætning til sådanne sprog baserer AgentC sig dog ikke på en meget specialiseret formel logik, men ligner i mange tilfælde mere et traditionelt proceduralt programmeringssprog.

Nøgleord: Intelligente agenter, programmeringssprog, oversættere, agentprogrammer, objekt-orienteret programmering.

Preface

This document has been produced as part of a Master's thesis which was carried out at the section of Computer Science and Engineering (CSE) of the Institute of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU).

Work on the thesis has been supervised at the CSE by prof. Jørgen Fischer Nilsson and assoc. prof. Hans Bruun.

I would like to thank both Hans and Jørgen for their feedback, helpful suggestions and comments. I also would like to thank Petra Dalgaard (former librarian at the IMM library) for her efforts to obtain a copy of [32] on my behalf.

Henrik Lauritzen

DTU, Lyngby, September 2002

Typographical Conventions

A brief overview of the general typographical conventions employed in this document is given in the following. In addition to these conventions some specialised notation (which is not relevant to the entire document) will be introduced before its first use.

Normal Text

The document text is presented in the font used in this paragraph.

Emphasised text

Text which has been emphasised by the author *is presented in the font used here.*

Quoted Material

Quoted material is presented in one of two forms: Shorter quotations whose contents are directly integrated in the running text, are presented "*inline, such as this*".

> "*Longer quotations whose contents are not necessarily integrated in the running text are formatted like this paragraph*".

Footnotes

Footnotes are consecutively numbered for each chapter, and are placed directly in the running text, such as shown here[1].

Cross-references

As customary, cross-references are usually integrated in the running text, such as: See section 1.1 on page 25.

In many cases where a more cursory reference is given, the cross-reference is shown as a small distinctive symbol suffixed to the last word of the relevant text. The format of such symbols is described in the following.

References to Parts of the Document

Chapters, sections, subsections, etc. are all parts of the document, and they have been given a unique number and/or letter combination. A cross-reference to a part of the document is presented in subscript, such as here$_{1.1}$ (which incidentally means section 1.1 on page 25).

References to System Requirements

System requirements$_2$ are each given consecutive numbers. Cross-references to system requirements are presented in superscript, showing the unique number preceded by the symbol #. For example, a reference to system requirement 1 takes the form$^{\#1}$.

Terms and Definitions

Terms whose importance is deemed to be significant are *emphasised when first used*, and are given an entry in the index. If the meaning of a term is particular to this document, then the index entry corresponding to the definition will be formatted in **bold face**.

[1] This is a footnote

Acronyms

The acronyms which are used in this document, but which are not in common use outside of the fields of software and software agents, have been collected on the list of acronyms D on page 217.

Special Notation

Special entities which are found throughout the document, like `File`, `class` or `package` names, and snippets of Java `code` or AgentC `code` are visually formatted in the style just shown. Larger program listings of either Java or AgentC code are set off from the running text, may contain line numbers, and enhance the visual appearance of the keywords of the language; the largest examples of such listings are found in found in appendix D and appendix C respectively.

Names of trademarks, languages, software programs and the like (Java and AgentC used in the preceding paragraph are examples of such names), are formatted in a particular typeface regardless of their surroundings. The particular visual format of such a name will be the normal document font, unless a particular visual format has been used for the name since its first origin. Java is an example of a name belonging to former category, while LaTeX 2$_\varepsilon$ — which incidentally was used to typeset this document — and many of the names invented for this thesis — among these most notably AgentC and Haplomacy — are examples of names belonging to the latter category.

Document Structure

The document has been divided into six parts, each having somewhat distinct subjects. The contents of each document part are briefly summarised in the following.

Part I: Problem Analysis

The text in this part is intended to be read sequentially, since much of the terminology of the document is defined here. The text is divided into two chapters, the first of which provides a general introduction to the general problem area, in addition to identifying the particular problems which are to be addressed by this thesis. The second chapter defines requirements for the program system which was developed as part of the thesis. These requirements are individually numbered for ease of reference, and a summary of all system requirements is additionally provided in section 2.7.

Part II: System Design

The chapters in this part each describe the design of a separate part of the software system under development. The parts into which the system is divided were already identified in the requirements specification found in part I.

Part III: AgentC Language Reference

The text found in part III describes in detail the syntax and semantics of AgentC. The text is mainly intended to serve as a language reference manual, but hopefully parts of it will also serve to provide an introduction to the language.

Part IV: System Test

This part of the document introduces a test scenario for the software system. The purpose of the system test is threefold:

1. The system test will ensure with a reasonable certainty that the software system works as intended.
2. The test scenario contains a general-purpose *test bed* which provides a basis for a rather wide range of experiments with agent-based software; the test scenario may be considered one such experiment.
3. The way the test scenario is implemented documents how the software system as a whole can be used. This part of the document can therefore also serve as a user's guide to the software system — provided that the user is willing to make good use of the index.

Part V: Conclusion

This part provides a discussion of the work described in the document, identifies future work, and concludes by providing a brief summary of the results of the thesis.

Part VI: Appendices

Contains various listings for the document, including the list of references, and a fairly rich index for ease of reference.

Contents

Part I

Problem Analysis

Chapter 1

Introduction

The notion of an agent is one of the more recent additions to the field of Artificial Intelligence. Agents have been the subject of active research during the past two decades, and real-life applications of multi-agent systems are beginning to emerge in areas as diverse as process control, information management and electronic commerce and patient monitoring [22]. However, many issues regarding the use of agents are still unresolved, not the least of which is the problem of how to engineer software systems based on agents [37].

The following sections introduce some fundamental problems relating to the use of agents. Where applicable, forward references to system requirements related to the problem will be included. Such references are formatted using a footnote-like symbol; for example, $^{\#9}$ refers to system requirement 9.

1.1 What is an [Intelligent] Agent?

Contrary to what one might believe, the fundamental question of what makes an agent cannot easily be answered; to be more precise: the question is answered differently by different people.

It may be beneficial to distinguish between the terms *agent* and *intelligent agent*, to stress that the term *agent* is a very broad definition. For example, Shoham [31] states that "*An agent is an entity whose state is viewed* *as consisting of mental components such as beliefs, capabilities, choices, and commitments*", then proceeds to remark that "anything *can be so described.*" Wooldridge [36] uses a slightly stronger definition, namely that "...agents *are simply computer systems that are capable of autonomous action in some environment in order to meet their design objectives*", restricting the notion of an *intelligent agent* to "...*one that is capable of* flexible *autonomous action in order to meet its design objectives*", using the following definition of the term *flexible*:

> - reactivity*: intelligent agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives;*
> - pro-activeness*: intelligent agents are able to exhibit goal-directed behaviour by* taking the initiative *in order to satisfy their design objectives;*
> - social ability*: intelligent agents are capable of interacting with other agents (and possibly humans) in order to satisfy their design objectives.*

The above definition corresponds to the *weak notion of agency* given in [35]. In the rest of this document the definition of an intelligent agent denotes a computer system which, in addition to the definition just given, is conceptualised and/or implemented in terms of *mentalistic* notions such as beliefs, capabilities etc. This corresponds to the stronger notion of agency described in [35].

For the sake of brevity, the term *intelligent agent* will be replaced by the shorter term *agent* in the following. No ambiguity should arise from this usage, as the loose definition of an *agent* will not used further.

The term *multi-agent system* is used to denote a computer system whose operation relies on the existence of at least two intercommunicating agents, whereas the weaker definition *agent-based system* is used to denote computer systems which merely contain or service one or more possibly intercommunicating agents.

The term *environment* denotes the parts of an agent-based system which are directly perceived or acted upon by agents. The subset of the environment which is comprised solely of agents is denoted the *social environment*, while the remains of the environment are denoted the *physical environment*. Although the interaction between an agent and its social environment can be standardised through the use of ACLs$^{\#8}_{1.3.3}$, the way in which the agent(s) interact with the physical environment$^{\#5}$ will necessarily be highly application-specific, because the physical environment itself is completely application-specific.

1.2 Why use Agents?

One of the most prominent reasons that agents are studied and used is that the notion of an agent is a *natural metaphor* [37]. This is by no means a recent idea. McCarthy [26] states that

> *"To ascribe certain* beliefs, knowledge, free will, intentions, consciousness, abilities *or* wants *to a machine or computer program is legitimate when such an ascription expresses the same information about the machine that it expresses about a person. It is useful when the ascription helps us understand the structure of the machine, its past or future behaviour, or how to repair or improve it. It is perhaps never logically required even for humans, but expressing reasonably briefly what is actually known about the state of the machine in a particular situation may require mental qualities or qualities isomorphic to them. Theories of belief, knowledge and wanting can be constructed for machines in a simpler setting than for humans, and later applied to humans. Ascription of mental qualities is most straightforward for machines of known structure such as thermostats and computer operating systems, but is most useful when applied to entities whose structure is incompletely known."*

The above paragraph is quoted in [35] as well as in [31], which furthermore mentions that "*. . . the gradual elimination of animistic explanations with the increase in knowledge is correlated very nicely with both developmental and evolutionary phenomena. In the evolution of science, theological notions were replaced over the centuries with mathematical ones.*"

While the above suggests why it may be a good idea to *explain* or *analyse* complex systems in terms of agents, this does not automatically justify why complex systems should be *implemented* or *engineered* in the same way (c.f. section 2.1). Indeed, [38] argues that "*given the relative immaturity of agent technology and the small number of deployed agent applications,* *there should be clear advantages to an agent-based solution before such an approach is even contemplated*", and "*There is certainly no scientific evidence to support the claim that agents offer any advance in software development — the evidence to date is purely anecdotal.*"

1.3 What is Required to Build an Agent?

The parts from which the agent is composed will in the following be denoted the *agent constituent*, while the structure of their composition is denoted the *intra-agent architecture*. The choice of intra-agent architecture is then an essential limiting factor of an agent's capabilities. Indeed, it is likely that different problems can only be solved by agent-based systems which employ different intra-agent architectures[#4].

The following agent constituents are identified as being necessary to build an agent as defined in section 1.1:

The *deliberative constituent* is responsible for the representation and maintenance of the agent's mental state, and thus realises the agent's pro-activeness and mentalistic aspects. In popular terms, the deliberative constituent can be described as the brain of the agent.

The *reactive constituent* is responsible for the agent's perception of the physical environment, as well as its reaction to the environment, which includes effectuation of any actions intended by the deliberative constituent. The reactive constituent thus covers the agent's autonomy and reactivity. In popular terms, the reactive constituent can be regarded as the motor, sensory and central nervous system of the agent.

The *communicative constituent* provides a means for the agent to interact with its social environment, thus covering the social ability. However, in this document the communicative constituent is viewed solely as a *medium* as well as a *protocol* for communication: only the reactive constituent can effectuate actions. Therefore, the communicative constituent will in popular terms correspond to the language and the air which is used to transfer verbal communication.

1.3.1 The Deliberative Constituent

The deliberative constituent has proven to be the most problematic constituent to realise in practice. Indeed, much effort (see e.g. [35, 36] for an overview) has been put into the question of developing logics which are computable as well as expressive enough to allow agent behaviour to be specified in terms of such logics. To date, two different approaches have been used with some success:

1. Specifying an agent in terms of temporal logic specifications, and directly executing these specifications [8]. While providing a formal framework for specification and verification of agents [9], this does however put serious restrictions on the choice of an intra-agent architecture.
2. Using a (restricted) first-order logical language in conjunction with modal operators in order to represent and to reason about the mental attitudes of the agent. Numerous examples exist: dMARS [3], is based on beliefs and intentions, using a temporal language to specify goals. AgentSpeak(L) [28] operates with beliefs, desires and intentions. PLACA [33, 32] adds plans to the beliefs, capabilities and commitments of Agent-0 [31, 30]. Interestingly, 3APL [19] allows an arbitrary (first order) logical language to be used for representation of beliefs.

While dMARS, AgentSpeak(L) and 3APL all have been formally specified, they do not directly provide the possibility of transforming an agent program into an implementation$^{\#12}$. The semantics of PLACA and Agent-0 has not been as strictly specified as the other languages just mentioned; on the other hand, they include an experimental interpreter [33, 34], which, however, enforces a specific intra-agent architecture$^{\#4}$.

1.3.2 The Reactive Constituent

In itself, implementation of a reactive constituent does not cause serious problems. However, when it comes to the balance between reactivity and deliberation, problems begin to occur. The question of how to make an agent's beliefs correspond to the reality of its environment is by no means easily answered [27]. Furthermore, the balance between reactivity and deliberation is not very well investigated, and few attempts have been made to address this problem: existing systems mainly focus on either of the two areas, instead of trying to unify them$^{\#5}$ [23]. For example, Agent-0 and PLACA assume a fixed execution cycle in which all received messages can be processed; such an assumption cannot be fulfilled in practice, where multiple concurrently executing agents often will be desirable, if not required.

1.3.3 The Communicative Constituent

Although the ability to communicate has been defined as synonymous with agency itself — "*An entity is a software agent if and only if it communicates correctly in an agent communication language*" [16] — such a definition is too narrow in practice: the ability to communicate is a necessary but not a sufficient condition when describing what an agent is$_{1.1}$. The importance of communication should not be belittled, however: "*It is because agents communicate that they can cooperate, coordinate their actions, carry out tasks jointly and so become truly social beings*" [7]. Additionally, agent-based communication can be used as a means of ensuring interoperability with legacy systems [16, 22], if a suitable intermediate agent communication layer is added to the system.

In contrast to almost every area involved in building multi-agent systems, the area of of inter-agent communication has actually been the subject of a substantial standardization effort. Already in 1993, the first standardised ACL, KQML, was proposed [18]. Although the standard has never officially progressed beyond the draft stage, quite much subsequent work, most notably [24], has been carried out in order to improve the original proposal of [18]. Furthermore, KQML has actually developed into a de-facto standard for inter-agent communication [38], and many toolkits providing KQML functionality are available; one example of such a toolkit is described in [1].

A more recent development has been undertaken by FIPA, the result of which is FIPA ACL. Surprisingly, FIPA ACL's syntax resembles KQML to a very high degree, but the language has been more thoroughly and rigorously specified in terms of its semantics, although it is still at the experimental stage. As a notable feature, the specification of FIPA ACL has been split into many smaller specifications, each dealing with a separate area: The *inter-agent architecture* (the infrastructure required to exchange messages) is the subject of the *Abstract Architecture Specification* [10]; *Message structure* (the syntax of a single message) is treated in the *Message Structure Specification* [11]; Languages to encode *message contents* (the information which is transferred in a message) are described in the *Content Language Library Specification* [13] and the documents referenced therein;

message semantics (rules for the use of different message types, or *performatives*) are treated in the *Communicative Act Library Specification* [12]; finally, *message protocols* (rules for sequences of messages) are specified in the *Interaction Protocol Library Specification* [14].

1.4 How do Agents and Objects Correlate?

Analogous to the question of what an agent is$_{1.1}$, the question of how to integrate agent-oriented and object-oriented programming paradigms is not easily answered.

In [31], Shoham proposes *Agent-Oriented Programming* as a new programming paradigm, which "*... specialises the framework by fixing the state (now called* mental state*) of the modules (now called* agents*) to consist of components such as beliefs (including beliefs about the world, about themselves, and about one another), capabilities and decisions, each of which enjoys a precisely defined syntax.*"

One of the key differences between the two concepts as stated in [37] is that "*the locus of control with respect to the decision about whether to execute an action is [thus] different in agent and object systems. In the object-oriented case, the decision lies with the object that invokes the method. In the agent case, the decision lies with the agent that receives the request*". Nevertheless, this differentiation is an academic one: the `State` pattern described in [15] is a simple example of how a purely object-oriented system may exhibit the behaviour used in [37] to characterise an agent-based system.

The preceding may lead one to conclude that the differences between agent- and object-oriented systems are small. This is not generally so: "*Put crudely, agents are more coarse-grained computational objects than are agents; they are typically assumed to have the computational resources of a UNIX process, or at least a Java thread. Agent systems implemented using object-oriented programming languages will typically contain many objects (perhaps millions), but will contain far fewer agents*" [37].

In the context of this document, as will be shown in the following$_{5.1}$, an agent is built from between two and five objects: the ACME$^{\#6}_{2.4.2}$, whose behaviour has been specified and implemented in terms of mentalistic attitudes; a component$^{\#5}_{2.4.1}$ which controls the ACME as well as the environment; and 3 extension modules$_{5.2}$, some or all of which could be combined in the aforementioned component.

Chapter 2

Requirements Specification

The following chapter contains the requirements specification for the program system whose construction is the subject of this thesis. The program system is named ACT[1]; in the following, ACT and *the system* will be used interchangeably.

As already mentioned in the introduction on page 25, system requirement x is referred to by the symbol$^{\#x}$.

2.1 Design Objectives

The main purpose of this thesis is to provide a toolkit (the ACT) which facilitates implementation of agents, in a way such that these agents can be deployed in the widest possible range of agent-based systems; the ACT can then serve as a foundation on which agent-based systems of varying complexity can be built.

Furthermore, it is the intention that the ACT encourage experiments with agent-based systems, by aiding the design and construction of such experiments.

[1] an acronym for AgentC Toolkit, c.f. the list of acronyms on page 217 and requirement$^{\#9}$

With the above objectives the ACT will hopefully be useful both in the process of prototyping larger-scale agent-based systems, but also in the important educational application gaining a better understanding of how best to solve problems in an agent-oriented way.

2.2 Design Considerations

In order to be of any value, the system must provide a suitable range of substantial components; failing this, the system cannot sufficiently reduce the workload of building an agent-based system$_{2.1}$. On the other hand, the components of the system must be general-purpose; if not, the system cannot be useful in a wide range of applications$_{2.1}$.

The goal of providing substantial yet general-purpose components cannot easily be achieved in practice, however, as the time available to the construction of the system is limited. The ACT will therefore necessarily exhibit a compromise between the two demands. The way the compromise falls out in practice may well prove to be the Achilles' heel of the system, since only a careful balance between the two demands will ensure that the design objectives are met.

The design requirements and considerations described in the preceding lead to a design which divides the ACT into three main *components*. The remaining sections define these components and provides a discussion of the requirements for each of these components affect the balance between substance and flexibility.

2.3 General Design Requirements

Requirement #1: A program system which fulfils the design objectives stated in section 2.1 shall be designed, implemented and made available as part of this thesis. ■

Requirement #2: The system shall have the form of a toolkit. Where possible, the components of this toolkit should be generic components which provide the basic functionality, but which also lend themselves to further extensions and specializations. ■

Requirement #3: The Java Programming Language shall be used to implement the ACT, in order to allow for maximal portability. By doing so it is ensured that the widest possible range of agent-based systems can be built$_{2.1}$. ■

2.4 The Agent Foundation Classes

So far$_{2.1}$, the word *toolkit* has been used without giving a precise definition of what it should be taken to mean. This is intentional, as the word is used with different meanings in this document.

When used as the final letter in the acronym ACT, *toolkit* should be taken to mean *a collection of software tools or applications, which each address a well-defined part of a common problem domain.*

The fundamental component of the ACT being the subject of the current section is a *toolkit* named the *Agent Foundation Classes*, denoted AFC. Here, *toolkit* assumes the traditional OOP terminology of [15]: "*A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. . . . Toolkits don't impose a particular design on your application; they just provide functionality that can help your application to do its job*".

In the remains of this document, the words *toolkit*, *package*, *class*, *interface*, *method* and *field* will be used exclusively in accordance with their mainstream use in OOP terminology in general and [15] and [17] in particular.

Requirement #4: The system shall contain a substantial class library (the AFC), which contains the components necessary to produce an ACME$_{2.4.2}$. The design should ensure that the components are independent of a specific intra-agent architecture. ■

2.4.1 The Reactive Constituent

No reactive constituent will be provided by the AFC, and it is furthermore assumed that *the intra-agent architecture is delivered by the reactive constituent.* In this way, it is ensured that no specific intra-agent architecture is enforced$^{\#4}$, but the decision may seem to be radical. However, as will be argued in the following, the reactive constituent will inadvertently consist almost entirely of application-specific components, which necessarily will have to be provided by the user in any case.

The main task of the reactive constituent is to provide a *control loop*, which in abstract terms can be defined by:

1 Receive input.
2 Process input.
3 Execute reaction.

Input to the agent consists of a number of *events*. An event is either a *message* originating from the social environment and received by the communicative constituent$^{\#8}$; an *internal event*, which is generated by and effects the deliberative constituent$^{\#6}$ alone; or an *external event*, that is, an event originating from the physical environment. The two former kinds of events are being handled by components provided by the AFC, and so the reactive constituent will only need to deal with the latter kind of events. As it is likely that even the simplest application will need very specific code in this regard, it makes good sense to give the reactive constituent complete freedom to provide this code.

The way messages and external events are handled is a task of the deliberative constituent$^{\#6}$. Still, most time-critical applications, or applications which will employ high-level communication protocols along the lines of [14], will most likely need to pre-process events. In the time-critical case, such pre-processing will be application specific (and probably mission critical), and is therefore best provided by the programmer who implements the specific system. Otherwise, the absence of a specific event pre-processing procedure provides maximal flexibility to design such a procedure, should the need arise. Thus, event pre-processing is essentially a hook through which the capabilities of the agent can be extended beyond the (necessarily limited) capabilities provided by the ACT.

The reaction to an event may be any combination of *internal actions*, which are carried out entirely within the deliberative constituent$^{\#6}$; *communicative actions*, which are carried out by the communicative constituent$^{\#8}$ in cooperation with the social environment; or *external actions*, whose meaning and execution depends solely on the physical environment. Again, only the latter possibility needs special care by the reactive constituent, and yet again the necessary code will be highly application-specific.

Requirement #5: The contents of the AFC shall be designed in such a way that a reactive constituent has freedom to receive, discard or produce messages; to receive, discard or produce external events; to modify (pre-process) or reorder received events; to inspect or modify the state of the deliberative constituent, and to allocate computing resources to the deliberative constituent. In this way, independence of a specific intra-agent architecture[#4] is ensured.

To ensure that the pre-processing of events can be fully effective, the reactive constituent should be allowed full access to inspect and even modify the state of the deliberative constituent, if so desired.

Finally, the reactive constituent should be free to choose and interpret the possible external actions. This will require that the deliberative constituent should be able to represent and reason about arbitrary external actions. ■

2.4.2 The Deliberative Constituent

To make up for the lack of reactive constituent[#5], the AFC should contain a substantial class library which provides the requisites necessary to directly build a deliberative constituent. In order to do so, it will be necessary to decide on a specific design of the deliberative constituent itself. The specific design of deliberative constituent provided by the ACT is denoted an AgentC[#9] *Mental Engine*, written ACME.

Requirement #6: The AFC shall allow for construction of an ACME, which essentially is a *mental state machine.* The *mental state* is specified in mentalistic notions such as beliefs, capabilities etc.; the *inputs* are the possible messages that can be received (in addition to any direct manipulation of the mental state[#5]); the *transition relation* or *behaviour* is specified by the ACME programmer and is realised through the rules$_{15}$ of the AgentC language[#9]; the *outputs* are the communicative actions and external actions which are executed as the result of a state transition. ■

2.4.3 The Communicative Constituent

The ACL should ideally provide a communicative constituent which is flexible enough to allow the agent[s] built from it to be used in varying inter-agent architectures. On the other hand, the communicative constituent provided should be substantial enough to allow for applications which directly use the communicative constituent provided by the AFC.

While providing an implementation of a general-purpose, standardised ACL such as KQML or FIPA ACL would fulfil these requirements, such an implementation will not be provided for the following reasons:

- It is not evident whether to prefer the old but de facto KQML standard [18, 24] over the recent but yet tentative FIPA ACL standard [12, 11].
- It is already possible to obtain various specific KQML implementations such as e.g. [1], and it will most likely be possible to obtain specific FIPA ACL implementations in the near future.
- Providing support for a full-scale ACL will not be feasible nor even desirable, since such functionality can be obtained elsewhere.

With these issues in mind, the requirements for the communicative constituent provided by the ACT are refined to the following:

Requirement #7: The ACT shall provide a simple *Generic Message Interface*, written GMI, in lieu of support for a full-scale ACL. The requirements for the GMI are the following:

- The GMI shall provide a message structure which is flexible enough to contain any message which is valid KQML or FIPA ACL. The GMI will therefore merely serve as an *abstract* ACL, which provides a means of passing information without assuming any semantics of this information.
- The GMI message structure shall allow a systematic translation between GMI messages and messages originating from a specific implementation of KQML, FIPA ACL or any similar ACL. Any message from such a concrete ACL should be convertible to an GMI message, whereas the reverse conversion is not necessarily possible since the contents of an GMI message are nearly unrestricted.
- Within the components provided by the AFC, most notably the ACME, it is required that the GMI is used exclusively for communication. The reactive constituent, which is application-specific, will then manage the conversion of GMI messages to and from any concrete ACL being employed, if necessary.

  As any outgoing GMI message's will have been generated by the reactive constituent itself or by the ACME it controls, the only requirement necessary in order to guarantee that these messages can be converted to a concrete ACL is that the ACME does not generate messages which cannot be converted; since the format of messages generated by the ACME is defined by the user, this requirement can only be enforced by the user.

■

With the preceding requirements$^{\#7}$, it is ensured that no specific intra-agent architecture is enforced$^{\#4}$, at the cost of a substantial communicative constituent. However, an extra requirement addresses this problem:

Requirement #8: The ACT shall provide the necessary generic components to allow a communicative constituent based on GMI communication to be built directly. ■

2.5 The AgentC

As already specified$^{\#6}$, the AFC *provides the requisites necessary to directly build an* ACME. These requisites, however, must be supplemented with tools which are sufficiently powerful, lest the ACT will not be able to *facilitate implementation of agents*$_{2.1}$. To address this problem a very important requirement is made:

Requirement #9: The system shall provide a high-level Agent Programming Language, which is able to express the *conduct* of an ACME. This specification language is denoted the AgentC. ■

The design of the AgentC will have a great impact on the range of ACMEs whose conduct can be specified, and hence the range of agent-based systems which can be built by the ACT will necessarily be limited by the design. However, due to the nature of this thesis and the issue at hand being quite complex, it is to be expected that the result will be somewhat simplistic. Rather than designing the AgentC from scratch, it makes good sense to borrow from existing APLs in this regard.

Requirement #10: In order to keep things simple, AgentC shall be based on the basic ideas of Agent-0 [31, 30, 34]. As a means of preserving generality, AgentC should not be based on a specific underlying logic and axiomatic system, but rather on a general — and simple — logical language. In this way it is ensured that a wide range of experiments with the design of agent-based systems can be conducted$_{2.1}$, and these experiments will furthermore serve to identify the areas where more complexity is needed.

In order not to oversimplify matters, AgentC should incorporate as wide a range of the features of PLACA [32, 33] (most notably planning capabilities) as possible. ■

The choice of language just presented$^{\#10}$ may seem an arbitrary one; furthermore, the choice of an old APL like Agent-0[2] may seem a dubious idea. Surprisingly, however, one of the conclusions of [20] is that "Agent-0, AgentSpeak(L), *ConGolog and* 3APL *form a close family of related languages*", indicating that the choice of a specific APL perhaps is of less importance than one might think. Starting from a simple language like Agent-0 and extending it to a more complex language like PLACA makes it possible to provide a working language design even under tight time constraints, but still makes it possible to refine the language at a later time.

What remains to be shown is that AgentC actually provides something which is not already provided by an existing APL. To this end, an extra, important requirement is given:

Requirement #11: AgentC shall be designed specifically to allow a specification to be *compiled* into an ACME, which exhibits the specified conduct, i.e., which directly executes the given agent program. Furthermore, the preceding requirements ensure that an ACME is *minimal* in terms of the amount of code whose availability must be assumed or whose mode of operation must be known, in order to integrate the ACME into an application. ■

In contrast to the existing more widely known, non-commercial APLs, the use of compilation in AgentC is new: 3APL is an abstract APL, which cannot be directly compiled, interpreted or executed. AgentSpeak(L) and Concurrent MetateM operate by directly executing a logic specification; doing so, the capabilities of agents constructed in the language are necessarily limited by the expressiveness of the language — a limitation which can be circumvented when using AgentC$^{\#5}$. Finally, languages like Agent-0, Agent-K and PLACA all rely on the existence of a specific interpreter, which again limits the range of applications in which the language can be used.

[2] Agent-0 was among the very first APLs to be suggested

2.6 The AgentC Compiler

Requirement #12: In order to succeed in facilitating implementation of agents$_{2.1}$, it will be necessary to provide an automated tool which produces a working agent implementation directly from a high-level specification. To this end, the ACT shall provide an AgentC *compiler*, denoted the ACC. ■

Requirement #13: The ACC shall take any valid AgentC specification as input, and as output produce an ACME whose behaviour is fully consistent with the given specification. The ACME which is produced shall be in the form of the source code for a single Java class which depends only on the classes found in the Java API and in the AFC. ■

By producing source code rather than byte code, it will be slightly less convenient to use the ACC in many cases; on the other hand, the source code will be an invaluable tool when validating, debugging or manually modifying the ACME produced by the ACC. Moreover, valuable time is saved because the ACC can rely on the Java compiler to perform basic optimizations, rather than having to handle these optimizations itself.

2.7 Summary of System Requirements

1 A software system which fulfils the design objectives stated in section 2.1 shall be designed, implemented and made available as part of this thesis.

2 The the system shall be provided in form of a toolkit (the ACT) whose contents should preferably be as generic and reusable as possible.

3 The ACT is to be implemented in the Java programming language.

4 The ACT shall contain a class library (the *Agent Foundation Classes*, AFC), which is sufficiently general to ensure independence of a specific intra-agent architecture.

5 The components of the AFC shall be designed in such a way that the reactive constituent has maximal freedom to handle events and to directly access the deliberative constituent, if necessary.

6 The contents of the AFC shall be substantial enough to allow a deliberative constituent (AgentC Mental Engine, ACME) to be built directly.

7 The AFC shall provide an abstract ACL (Generic Message Interface, GMI), which enables transport of arbitrary messages.

8 The ACT shall provide generic components suitable for construction of a communicative constituent based on GMI.

9 The system shall contain an APL, AgentC, which can express the conduct of a single agent (ACME).

10 The syntax of AgentC will be based on Agent-0 and the extensions found in other languages, most notably PLACA.

11 AgentC shall be designed specifically to allow for compilation of an AgentC specification into an ACME program which exhibits the specified behaviour.

12 The ACT shall include a compiler (AgentC Compiler, ACC) among its contents.

13 Output from the ACC shall be in the form of Java source code.

Part II

System Design

Chapter 3

Organisation of the ACT

The ACT has been organised into four different packages, all of which are placed under the common namespace `dk.dtu.imm.cse.agent.act`. For the sake of readability, the package name prefix has been omitted from class names in the rest of this document part.

The packages of the ACT are the following:

`acc` contains the ACC implementation. A more detailed description is found in chapter 7.

`afc` contains the classes of the AFC. A detailed description is found in chapter 5.

`demo` contains an example application based on the components of the ACT. The example application is described in chapter 19.

`testbed` contains the implementation of a standardised test bed application which can be used as the basis of experiments with the design of agents. The contents of the package are described in chapter 18.

`util` contains various generic classes. The contents of the package are described in chapter 8.

In addition to the above, the AGENTC language is itself an intangible part of the ACT. A description of the general design considerations can be found in chapter 6; a full language reference is the subject of part III.

The following chapters contain high-level descriptions of the most important aspects of the contents of the ACT. A full API reference manual is provided in electronic form on the companion CD found in appendix D.

Chapter 4

The Generic Message Interface

The system requirements$^{\#7}$ specify that a *Generic Message Interface* should be included in the ACT, and that the GMI message structure should be sufficiently general to encode any message originating from a specific ACL like KQML or FIPA ACL. The implication of the requirement is that it is not possible to assume or define any semantics of the message content language.

In recognition of the above it has been decided that the GMI message structure should be a general `Map` data structure, with the only restriction that the key values should be `String` instances. A communicative constituent is then free to decide the set of keys it will recognise, and no restrictions are placed on the values which can be stored under these keys. Hence it is ensured that the communicative constituent is free to choose how translate between GMI and the specific ACL employed — if the GMI is not used directly. Indeed, the GMI message structure should be sufficient for any application where interoperability with other agent-based systems is not required.

The ACT includes a generic communicative constituent based on GMI$_{8.1}$.

Chapter 5

Design of the AFC

The contents of the AFC have been kept to a minimum. Essentially, the AFC contains only two classes (in addition to the components necessary to construct them), namely `afc.Acme` and `afc.AcmeKnowledgeBase`. These are described in the following sections.

5.1 The Abstract ACME

The class `afc.Acme` is the `abstract` superclass of all ACMEs generated by the ACC. The UML diagram in figure 5.1 on the next page shows which classes and interfaces the ACME relies on; all of these are described in the following.

A brief account of the methods provided by the class `afc.Acme` itself is found below.

public **final** `Object getId`();

Returns the unique object used for identification of the ACME. The object must be provided to the constructor.

public **final** `Actuator getActuator`();
public **final** `Investigator getInvestigator`();
public **final** `Messenger getMessenger`();

Accessor methods to obtain the *extension modules*$_{5.2}$.

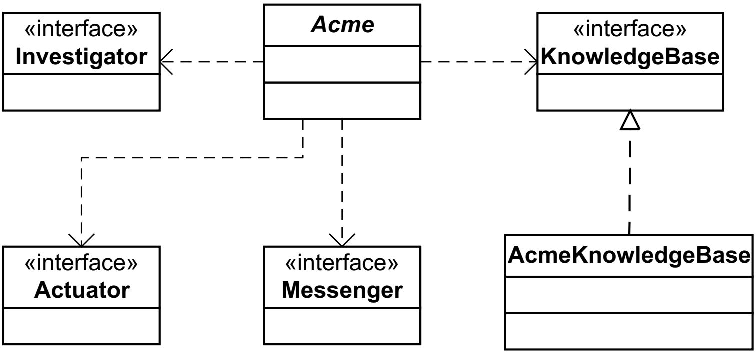

Figure 5.1: ACME Relationship Diagram

public **final** `KnowledgeBase getKnowledgeBase`();

Retrieves the knowledge base. The method makes it possible to modify the knowledge base outside of the ACME, e.g. by the reactive constituent$^{\#5}$.

public **abstract** **int** `getMaxAttitude`();

Determines the maximal attitude which the knowledge base can hold. The method is generated by the ACC$_{13.2}$.

protected **abstract** **void** `initKnowledgeBase`(`KnowledgeBase kb`);

Initialises the given knowledge base to its initial state. The method is produced by the ACC$_{13.3}$.

protected `KnowledgeBase createKnowledgeBase`();

Creates a new knowledge base (used in `init()`). The default implementation returns a new `AcmeKnowledgeBase`$_{5.3}$ instance.

public **void** `init`(`Actuator`, `Messenger`, `Investigator`);

Initialises the knowledge base and sets the extension modules.

public **int** `getAchievementId`();

Specifies the achievement attitude$_{15.1.1}$. The default implementation returns `0`.

public **boolean** `doAction`(`String`, `Object`[]);
public **boolean** `xeqAction`(`String`, `Object`[]);

Executes the specified action, delegating to the ACMEs actuator; the former method also updates the knowledge base on a successful operation, using the achievement attitude$_{15.1.1}$. The methods are used to execute an AgentC action statement$_{15.1}$.

public `Object` `query`(`String`, `Object`[]);

Delegates to the investigator to perform the given query. The method is used to execute an AgentC query$_{14.6}$.

public **void** `send`(`Map`);

Sends the given GMI message, using the ACMEs messenger. The method is used to execute an AgentC `SAY`-statementsec-sayStm.

public **final** **boolean** `isEqual`(`Object`, `Object`);

A utility method used in the generated code to compare two objects for equality. The method is used in order to evaluate for equality as a Java expression rather than generating temporary variables, because each operand needs to be referred twice.

5.2 The ACME Extension Modules

The `Actuator`, `Investigator` and `Messenger` interfaces are collectively known as *extension modules*. Their prime function is to ensure independence of a specific intra-agent architecture$^{\#4}$, by providing an abstract interface between the ACME and its environment.

The user can freely choose whether or not to implement the extension modules into a single class (which could also make up the reactive constituent, if desired.), and whether or not to share these extension modules between more than one ACME (if proper synchronization is provided in the implementation).

5.2.1 The Actuator

The *actuator* serves as the interface between the ACME and the physical environment. The interface specifies one method:

```
public boolean xeq(String, Object[]);
```

The interpretation of the parameter values is determined by the specific actuator$^{\#5}$. The return value is required to be `true` if and only if

1. the `String` is identified as a known action.
2. the `Object[]` action parameters have the required number and run-time types.
3. the specified action could be effectuated in the physical environment of the agent.

5.2.2 The Messenger

The *messenger* serves as the interface between the ACME and the social environment, but not vice-versa[1]. The interface specifies one method:

```
public void send(java.util.Map);
```

The `Map` instance contains a GMI_4 message. The messenger is free to choose how to obtain an addressee from the properties of the provided message.

5.2.3 The Investigator

The *investigator* serves as an interface between the ACME and the Java program in which it runs. The module is provided in order to allow the ACME to calculate or access values which AgentC cannot directly express or obtain. The investigator also provides a way to optimise critical parts of the ACME code, if desired, since there is no restriction on the code which can be performed by the investigator.

The interface specifies one method:

```
public Object query(String, Object[]);
```

[1] The reactive constituent is responsible for retrieving messages and handling these (typically by calling a specific procedure in the ACME).

The investigator implementation is free to determine what operations to perform and which value to return for any combination of parameters.

5.3 The Knowledge Base

The *knowledge base* is an intrinsic part of the ACME — it contains the mental state of the ACME, which, in the normal case, is the only internal state maintained by the ACME[2].

The range of values which can be stored in the knowledge base must be carefully chosen, because this range of values effectively determines the logical foundation of AgentC, and thus limits the usability of the language.

As the requirements$^{\#10}$ have already stated, it has been decided to use a simple, but generic logical representation in AgentC. This decision directly opposes what is the customary way of designing ACLs: Agent-0, for example, is founded on a logic system based on choice and commitments; the specification of properties, assumptions and axioms related to that specific logic (which never was very strictly formalised) fills up 8 pages in [31]. According to [33], PLACA operates with a mental state of beliefs, capabilities, plans and intentions, for which a formal specification is supposedly the main topic of [32]. AgentSpeak(L) [28] and its successor dMARS [3] are based on a logic of beliefs, desires, and intentions (a so-called BDI architecture), which is the subject of a large — and still growing — quantity of articles.

The benefits of using a simple logical system are many, however. First of all, it allows the construction of a working implementation which can be refined at a later time, rather than producing a nonworking implementation which has only theoretical interest. Second, a simple underlying logic drastically reduces the computational resources required to execute an agent program. Third, by allowing the user to determine which mental attitudes to be used, rather than enforcing a specific set of these attitudes, the system is more suitable for experiments$_{2.1}$, and for prototyping larger agent-based systems$_{2.1}$: although the BDI architecture is predominant in the agent literature, and even in some applications, it is far from certain that BDI logics are the best choice for agent construction in the general

[2] The extension modules are allowed to maintain and modify an internal state, if required, but normally they will not need to do so.

case (although it is almost certain that such logics are not the best choice in *every* case).

5.3.1 Facts

As a starting point for the implementation, the simplest possible knowledge base has been chosen: The knowledge base is a data structure which holds a set of *facts*.

The facts used in the system correspond to DATALOG *ground facts* (c.f. [29] §6.2.4), with the addition of a *mental attitude*, simply written *attitude*. An attitude is a nonnegative integer value, which the AgentC programmer is free to choose$_{13.2}$. A fact therefore consists of 3 components: an attitude, a *predicate symbol* (a string), and a list of *terms*, which are simple values$_{11.1}$.

The AFC includes a class `Fact`, which is used to represent the facts just defined. The class uses the types `int`, `String` and `Object[]` to represent the attitude, predicate symbol and list of terms, respectively. The main function of the class is to encapsulate these 3 values into a single Java object, which can be transferred in a message[3]; to provide the necessary code to *match*$_{5.3.1.1}$ such values against each other; and to compare them for equality.

5.3.1.1 Terminology of Facts

Consider two arbitrary facts $\alpha\ \pi_\alpha\ \tau_{\alpha_0}, \ldots, \tau_{\alpha_n}$ and $\beta\ \pi_\beta\ \tau_{\beta_0}, \ldots, \tau_{\beta_m}$.

The *arity* of a fact is equal to the number of terms; hence, the facts above have arities n and m, respectively.

The facts are *similar* iff $\alpha = \beta \wedge \pi_\alpha = \pi_\beta$.

The facts are *compatible* iff they are similar and additionally $n = m$.

Consider two compatible facts $\alpha\ \pi\ \tau_{\alpha_0}, \ldots, \tau_{\alpha_n}$ and $\alpha\ \pi\ \tau_{\beta_0}, \ldots, \tau_{\beta_n}$. Let $\mu \subseteq \{0, \ldots, n\}$. Then, the facts are said to *match with regards to* μ iff

$$\forall i : i \geq 0 \wedge i \leq n \Rightarrow i \in \mu \vee \tau_{alpha_i} = \tau_{beta_i}$$

Two facts are *equal* iff they are compatible and match with regards to the empty set.

[3] The `Fact` is serializable for the same reason.

5.3.2 Knowledge Base Operations

The `KnowledgeBase` interface specifies a number of operations which are necessary for an AGENTC program to be run. For efficiency, all operations are specified in two variants: whenever an operation requires a `Fact` as a parameter, a corresponding method requiring three parameters of types `int`, `String` and `Object[]`, is also specified. The knowledge base implementation can easily implement one of the operations in terms of the other, either by specifying

```
... operation(int attitude, String psymb, Object[] terms) {
    return operation(new Fact(attitude, psymb, terms));
}
```

or[4]

```
... operation(Fact f) {
    return operation(f.getAttitude(), f.getName(), f.getTermList());
}
```

The benefit of specifying the two variants is that a new `Fact` instance may not be necessary in some cases. The ACC, for example, avoids explicitly creating `Fact` instances if possible.

```
public boolean clear();
public boolean isEmpty();
public int size();
public Iterator iterator();
```

General methods to maintain and investigate the contents of the knowledge base.

```
public Object getLock();
```

Returns the object on which the operations of the knowledge base are synchronised.

```
public boolean add(Fact);
```

[4] Because the `Fact` assumes immutability of its terms and internally caches its hash code for efficiency, the `getTermList` method, which directly returns the `Object[]` instance used by the `Fact`, is package-private. In some implementations, therefore, it will be necessary to retreive the list of terms in another way, e.g. by using the less efficient `getTerms` method.

public **boolean** `add`(**int**, `String`, `Object`[]);
Ensures that the given fact is found in the knowledge base; returns true if the fact was not present before the addition.

public **boolean** `contains`(`Fact`);
public **boolean** `contains`(**int**, `String`, `Object`[]);
Queries whether a given fact is present in the knowledge base.

public `List` `match`(`Fact`, `BitSet`);
public `List` `match`(**int**, `String`, `Object`[], `BitSet`);
Returns the facts in the knowledge base which match the given fact with regards to the given set[5].

public **boolean** `remove`(`Fact`);
public **boolean** `remove`(**int**, `String`, `Object`[]);
Removes a fact from the knowledge base.

public **int** `remove`(`Fact`, `BitSet`);
public **int** `remove`(**int**, `String`, `Object`[], `BitSet`);
Removes all the facts from the knowledge base which match the given fact with regards to the given set.

5.3.3 Implementation of the Knowledge Base

The AFC contains one implementation of the `KnowledgeBase` interface, namely the class `AcmeKnowledgeBase`. The implementation has been designed with the following requirements:

- The running time of the operations should be moderate, even for a large knowledge base.
- The memory requirements for even a large knowledge base should be moderate.
- All access to the knowledge base should be synchronised, such that the reactive constituent can concurrently inspect or modify the state of the knowledge base[#5], if required.

[5] The `BitSet` has been chosen because it is the most memory-efficient way of storing a set of relatively small integers — memory consumption is $8 \cdot \lceil \frac{n}{64} \rceil$ when the largest number stored in the set is n. Due to the highly optimised `nextSetBit` routine provided by the `BitSet`, the added penalty to iteration performance is negligible.

5.3.3.1 Structure of the `AcmeKnowledgeBase`

The structure of the `AcmeKnowledgeBase` has been dictated by the requirements to keep both running time and memory consumption low. The key idea behind the design is that all operations (except iteration, which is not required for an AgentC program anyway) are dependent on the ability to quickly locate facts which are compatible with another fact. Therefore, the `AcmeKnowledgeBase` has been structured such that similar facts are placed into their own separate container, which can be queried or modified independently of the remains of the knowledge base[6]

Since similar facts are grouped together, it is possible to reduce memory consumption by eliminating redundant attitudes and predicate symbols, a strategy which is used by the `AcmeKnowledgeBase`. Hence the knowledge base does not store any `Fact` instances internally, but only lists of terms. As already described, this choice may also save a few `Fact` instances to be created when the knowledge base is merely queried; however, a set of new `Fact` instances will necessarily be instantiated as the result of a `match`.

Figure 5.2 shows the structure of a knowledge base which has been initialised by the AgentC code listed here (please refer to part III for a detailed explanation):

```
ATTITUDES {
    #BELIEVE = 1;
    #INTEND = 2;
}
FACTS {
    #INTEND i(”a”, ”b”);
    #INTEND i(”d”, ”c”);
    #BELIEVE b1(”a”, ”b”, ”c”);
    #BELIEVE b1(”d”, ”e”, ”f”);
    #BELIEVE b2(”A”, ”B”, ”C”);
    #BELIEVE b2(”D”, ”E”, ”F”);
    #BELIEVE b2(”G”, ”H”, ”I”);
}
```

[6] The `AcmeKnowledgeBase` groups similar facts rather than compatible facts as a compromise between memory consumption and running time. If the AgentC programmer does not use the same predicate symbol with term lists of different arities, however, the facts which are similar will coincide with the facts that are compatible, thus yielding optimal performance.

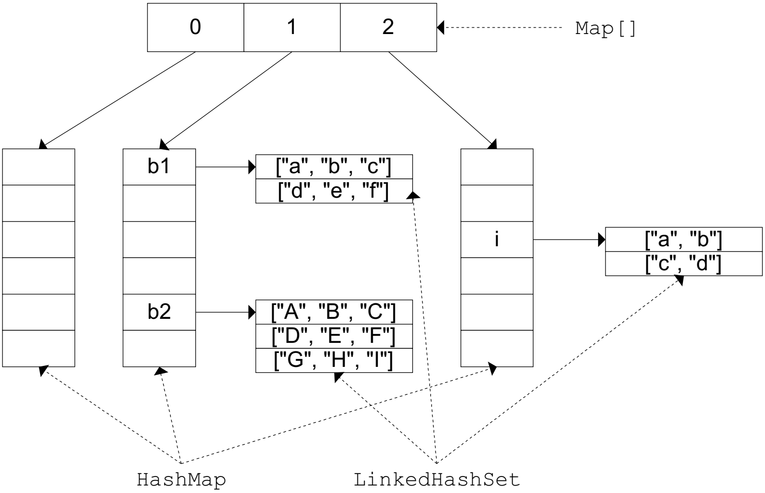

Figure 5.2: `AcmeKnowledgeBase` internal structure

5.3.3.2 Complexity of Knowledge Base Operations

Let $\alpha\ \pi\ \tau_0, \ldots, \tau_{n-1}$ be an arbitrary fact. The running time of any operation will asymptotically be the same regardless of whether the value is given as a `Fact` instance or as three values, since the three values can be extracted from the `Fact` instance in three constant-time operations.

Given the knowledge base structure shown in figure 5.2, a set of facts similar to another fact can be found in amortised constant time, since the operation involves one array indexation and a subsequent table lookup[7]. The asymptotical running time of any knowledge base operation will therefore not be affected by the time required to look up the set[8].

The `add`, `contains` operations, and the simple `remove` operation, will require $O(\max(1, n))$ running time, since the operations need $O(1)$ running time

[7] It is assumed here as well as in the following analysis, that strings can be considered as having a maximal size, such that the time required to compare two predicate symbols or terms for equality can be considered to be a constant. In general, however, the longer the string values involved, the lower performance will be.

[8] The constant times involved in iteration operations will generally be large. Iteration need not be very efficient, though, since the operation is not required for an AgentC program.

to locate the required set and a subsequent $O(n)$ `add`, `contains` or `remove` operation in that set.

The `match` and the extended `remove` operation require a $O(1)$ set lookup operation, and subsequently need to iterate over that set. Let σ be the size of the set; then σ iterations are necessary, and each iteration will cost $O(n)$ if the arity of the current list of terms is n, and $O(1)$ otherwise (i.e. if the facts under consideration are similar, but not compatible). The total running time therefore amounts to $O(\max(1, \sigma, n \cdot \sigma))$.

To restate the above in general terms, if the *size* of a `Fact` is defined as being the sum of

1 for each numeric value in a term.
n for each string value of length n, either in a term or in the predicate symbol.

then the worst-case running time of the simple knowledge base operations is therefore directly proportional to the size of the fact; for the matching knowledge base operations the worst case running time is proportional to the size of the fact multiplied by the number of similar facts existing in the knowledge base.

Chapter 6

Influential Factors in the Design of AgentC

To paraphrase the preceding design considerations, the key idea behind AgentC is to provide a general language which allows agent-oriented specifications to be compiled into traditional object-oriented programs, which can be seamlessly integrated into object-oriented applications.

Rather than removing itself from the OOP paradigm and reinventing everything in an agent-based way, AgentC adds a new layer of functionality to traditional OOP programs, a functionality which is rooted in (but not limited to) agent-oriented programming. The result is a *procedural* language which operates on a *mental state*, and which allows for *communication* with other programs.

6.1 The Knowledge Base

As already described in section 5.3, the knowledge base design has a large impact on the design of AgentC. Its capabilities in terms of logical expressiveness are admittedly simple. However, the present design is a starting point for a future more advanced language. Indeed, the simplicity of the present design has been influenced by the intention to use the ideas of

PLACA, which turned out to be impossible at a rather late stage in the design process (the footnote to [32] elaborates on this difficulty).

6.2 Fundamental Properties

Some fundamental properties of AgentC programs are partly borrowed from the ideas originally found in [31]: an Agent-0 program basically consists of a series of *commitment rules* — which in PLACA have become *mental change rules* — guarded by a *message condition* and a *mental condition*. The terminology of rules$_{15}$ and mental conditions$_{15.3.1.4}$ remains in AgentC, but due to its more procedural nature, rules make up a procedure$_{13.4}$ which is executed sequentially, and most rules can be recursively composed. An Agent-0 program, on the other hand, contains a single list of commitment rules, and depends on the interpreter to continuously check and execute these rules when their condition is fulfilled; the same scheme is carried over in PLACA.

6.2.1 Lexical Syntax

Overall, the lexical syntax of AgentC resembles that of Java, except that keywords consist of uppercase letters. Comments$_{10.4}$ and literals$_{10.9}$, for instance, are directly borrowed from the lexical syntax of Java.

6.2.1.1 Variables

The use of variables in AgentC — as well as their lexical syntax$_{10.6}$ — originates directly from Agent-0. In contrast to the original Agent-0 implementation[1], variables in AgentC's mental conditions are implicitly universally quantified — but the effect of implicit universal quantification in mental conditions result in AgentC behaving somewhat like PLACA as described in [33]: "*Every rule is fired once for each possible match*".

[1] Variables are normally existentially quantified in Agent-0. Universally quantified variables are described in [31] as a feature which "*was not included in is actual implementation*".

6.2.2 Keywords

The lexical syntax of AgentC shares two keywords with PLACA: `ADOPT`$_{15.6.1}$ and `DROP`$_{15.6.2}$. Otherwise, the syntax of AgentC does not resemble Agent-0 or PLACA: the keywords of these languages mostly relate to specific attitudes or actions (`DO, INFORM, REQUEST, BELIEVE, INTEND`) — which AgentC lets the user define$_{13.2,15.1}$.

6.3 Grammar

The syntax of AgentC bears a closer resemblance to that of Pascal than to Agent-0 or PLACA: AgentC does not require (or generally even allow) parentheses before a keyword, and is more readable for the same reason. Additionally, AgentC has the keywords `IF`$_{15.3}$, `ELSE`$_{15.3}$, `PROCEDURE`$_{13.4}$ and `SELF`$_{14.4}$ in common with Pascal, and its pattern matching syntax borrows from the syntax of SML.

6.4 Message Guards

The syntax of message guards$_{15.9}$ (corresponding to Agent-0's message conditions) has been inspired by the syntax used in Agent-K [2] (an integration of Agent-0 with KQML communication).

6.5 Memory of Past Actions

The `DO`-statement$_{15.1}$ of AgentC automatically records in the knowledge base the actions which has been performed, effectively allowing the AgentC program to 'remember' its past actions. This idea was originally found in the Elephant 2000 language [25], and persists as an axiom of the logic of PLACA, according to [33].

Chapter 7

Implementation of the ACC

With the establishment of the syntax of AgentC and the implementation of the AFC, implementation of the ACC is rather straightforward.

The parser and lexer of the ACC has been constructed using JavaCC [21], while the data structure to represent the abstract syntax tree is coded by hand (most of the classes of the `acc` package fall into this category).

Compilation is performed in three stages

1. Parse the input into an abstract syntax tree (using an instance of `acc.ParseTree` as the top-level container). Part III describes in details the syntax of AgentC, which is also listed in appendix A.
2. Do a semantic check on the procedures of the program, to verify that variables are properly used. At the same time variables are classified into *variable defs* and *variable uses* as described in chapter 12.
3. Generate code from the abstract syntax. The language reference found in Part III contains detailed descriptions of the code which results from compiling the abstract syntax.

7.1 Using the ACC

The purpose of the ACC is to translate AgentC into Java code, following the specifications in part III. What is missing from these specifications, however, is a description of how to run the ACC itself. Such a description is the subject of this section; the reader should be warned that some of the terminology used here is first defined in part III.

7.1.1 Running The ACC

The ACC is implemented in the class `acc.Acc`. As the remaining components of the ACT it requires a Java runtime environment of version 1.4 or later in order to run[1].

Since the `Acc` class implements the compiler routine as the method `public static void main(String[] args)`, the ACC can be invoked directly by the system command line

```
java dk.dtu.imm.cse.agent.act.acc.Acc
```

assuming that the ACT binary code is found somewhere in the class path. The file `act.jar` found on the CD enclosed with appendix D contains the ACT in binary form; it defines `Acc` as its main class, which allows use of the shorter command line

```
java -jar act.jar
```

Each contiguous character sequence following the above command line are known as *arguments*; the arguments whose first character is a hyphen (`-`), are known as *options*.

7.1.2 Choosing the Input

The location from where the ACC takes its input is determined from the arguments in the following way:

[1] The ACT depends on the new classes `java.util.LinkedHashSet` and `java.util.LinkedHashMap` found in Java 1.4, as well as the method `nextSetBit` which was added to `java.util.BitSet` in the same release. Otherwise the ACT only depends on components already found in Java 1.2.

- If at least one argument which is not (part of) an option is specified, then the input will be the *concatenation* of the files whose names were specified in these arguments[2].
- Otherwise input will be taken from `System.in`.

7.1.3 Controlling the Output

The output from the ACC is controlled in the following way:

- If the option `-o` has been specified, the following argument will be interpreted as a file path, and the resulting output will be written to that location[3].
- Otherwise the output will be written to `System.out`.

7.1.3.1 Choosing the Package Name

The generated output will by default not contain a package declaration. However, it will if a package name is specified as explained below:

- If the `-o` option is used, the specified path ends with the suffix `.java`, and the path contains at least one path separator character, then the package name will be the part of the path name which precedes the last path separator character, with the modification that all path separator characters are substituted by a 'dot' (`.`).
- A package name can optionally be specified as the argument following the option `-pck`.

7.1.3.2 Choosing the Class Name

The generated output will by default use the class name `AccOutput`. The class name to be used can be specified in two different ways:

[2] The files are concatenated in the order they are specified. In case of an error the line number in the error message will be a *logical* line number with reference to the whole input.

[3] In case of multiple `-o` options being given, then only the last of these will have any effect. If a file already exists in the specified location, it will be *overwritten* without further notice.

- If the `-o` option is used and the specified path ends with the suffix `.java`, then the class name will be a substring of the path delimited by the last occurrence of a path separator character (or the beginning of the path, if no such character is found), and the `.java` suffix.
- A class name can optionally be specified as the argument following the option `-cls`.

7.1.3.3 Specifying a Superclass

The superclass of the generated class can be specified as the argument following the `-ext` option (the full class name should be used). The ACC will verify that the the specified superclass is or descends from `afc.Acme`.

As a side-effect of defining a superclass, the generated code will *inherit* some properties of that class; section 7.1.4 describes how.

7.1.3.4 Specifying Interfaces

For each interface which should be implemented by the generated class, a `-impl` option should be given, followed by an argument containing the full path of the interface.

As a side-effect of defining an interface, the generated code will *inherit* some properties of that interface; section 7.1.4 describes how.

7.1.4 Inheritance in AgentC

The output from ACC is a Java class which inherits from its superclass and implemented interfaces in the normal way. However, the AgentC code from which it is created can also *inherit* from these entities in the sense that the ACC produces implicit AgentC code based on their contents, in the manner described in the following.

7.1.4.1 Fields

Fields which are `public`, `static` and `final` may result in a symbol definition$_{13.1}$ subject to the following conditions:

- If the field has a primitive type then
    - If the type of the field is `int` or `double`, then a new symbol definition using the field's value will be generated; when the AgentC program is compiled, the corresponding code will be produced as normal$_{13.1}$.
    - Otherwise the presence of the field will have no effect.
- Otherwise the name of the field will be defined as a symbol, but *no corresponding code will be generated*: any use of the symbol will refer directly to field inherited by the output class.

7.1.4.2 Methods

Inherited methods which have a suitable signature will result in a procedure requirement declaration$_{13.4}$ subject to the rules defined below.

A method signature is suitable for inheritance only if

1. The method is either `public` or `protected`.
2. The declared return type is `Object`
3. The number of declared parameters is at least 1.
4. The declared parameter type is `java.util.Map` for the first parameter and `Object` for every subsequent parameter.
5. The method does not throw any checked exceptions (c.f. [17] §11.2).

Each suitable method causes the ACC to do the following:

- If the method is `final` a procedure requirement declaration is generated, but *no procedure body is allowed to be specified.* No extra code will be generated, but any use of the corresponding procedure in the AgentC program will call the inherited method.
- Otherwise, if the method is `abstract`[4] then a normal procedure requirement declaration is produced.
- Otherwise the result is a procedure requirement declaration which does not require nor prohibit a procedure body to be declared.
    - If no body is subsequently declared, the generated code will refer to the method inherited by the output class.
    - Otherwise the generated code will override the inherited method.

[4] All methods inherited from an interface are `abstract` by definition.

Chapter 8

Generic Utilities

In accordance with the system requirements[#2], the ACT contains a collection of generic utilities, all of which are placed in the `util` package. The most important of these utilities are described here.

8.1 Generic Communication Components

As required[#8], the ACT includes generic components to allow a GMI communicative constituent to be built directly.

8.1.1 The `Mailbox`

The `Mailbox` class implements a FIFO queue for GMI messages (i.e. `Map` instances). The implementation is designed to be used by multiple threads, and provides functionality to suspend a thread which tries to receive a message from an empty mailbox, and to resume it once a new message becomes available.

8.1.2 The `PostOffice`

The `PostOffice` class is a data structure which contains a number of `Mailbox`es, indexed by a user ID (an `Object` instance). The `PostOffice` allows users to be registered and unregistered dynamically, and provides a facility to broadcast a message to all registered users.

8.1.3 The `DefaultMessenger`

The `DefaultMessenger` class uses a `PostOffice` to distribute messages to their recipients. The `DemoMessenger` extracts the addressee of a message under the key value `"to"`; if such a value does not exist, the message will instead be broadcast to all users known to the `PostOffice`. Before a message is sent, the sender (the ID of the ACME to which the `DefaultMessenger` is attached) will be automatically added to the message under the key value `"from"`, if such a value does not already exist.

8.1.4 The `MessageController`

The abstract class `MessageController` provides the basis for a communicative constituent which uses a `Mailbox` to receive incoming messages.

The `MessageController` is a `Thread` which continuously waits for new messages, and lets a method defined by a subclass handle the message at the time they are extracted from the `Mailbox` queue; the `MessageController` can optionally also generate 'empty' messages[1] after a given period has expired without real messages being received.

The functionality provided by the `MessageController` makes it well suited as the basis of a communicative constituent, a reactive constituent or (in simple cases) a combination of the two — the example application described in chapter 19 uses such a combination.

8.2 Generic ACME Extension Modules

The classes `GenericInvestigator` and `GenericActuator` are abstract investigator and actuator implementations. They implement the `query` and `xeq` methods, respectively, by using Java reflection to find an appropriate method in *the class which extends them.* By extending these classes,

[1] Technically, the value `null` is used to represent an empty message when the handler method is invoked.

the user needs only to specify the operations which are supported by the extension module, using Object as the type for each parameter.

As an example, consider the two Java classes

```
public class ExampleActuator extends GenericActuator {
    public void testAction(Object p1) {
        System.out.println(p1);
    }
}
```

and

```
public class ExampleInvestigator extends GenericInvestigator {
    public Object add(Object p1, Object p2) {
        return new Integer(((Number)p1).intValue() +
                ((Number)p2).intValue()));
    }
}
```

and assume that these are used as extension modules for an ACME constructed by the AgentC code

```
PROCEDURE example() {
    IF (XEQ testAction()) {
        RETURN −1;
    }
    ELSIF (XEQ testAction(”testParam”)) {
        RETURN Q add(4, 5);
    }
    ELSE {
        RETURN −2;
    }
}
```

The return value from the example method in the generated ACME would then be 9 (an Integer instance), and the string "testParam" would have been printed to System.out. The reason is that testAction() fails because no matching method is found in the ExcampleActuator; testAction("testParam") succeeds, however, because the ExampleActuator does specify a matching method in this case, and hence the result of the query Q add(4, 5) — for which the ExampleInvestigator specified an appropriate method — will be returned from the procedure.

Part III

AgentC Language Reference

Chapter 9

Notation

The current part of the document contains a reference manual to the AgentC language; the following sections introduce the special notation used in the language description. While the text provides an exhaustive language reference, explanations are kept short. For further details regarding how the ACC is implemented or about the contents of the AFC, please consult part II.

9.1 About Grammars

All syntactical aspects of the AgentC language are specified using the type of BNF grammar defined below:

9.1.1 Non-terminals

Non-terminals are presented in the normal document font, and are enclosed in angular brackets:

⟨Non-Terminal⟩

9.1.2 Terminals

Terminals are presented verbatim, in typewriter font:

`terminal`

9.1.3 Production Rules

Production rules contain a non-terminal on the left-hand side, the expansion on the right-hand side, and the symbol ::= in between:

$\langle$Non-Terminal$\rangle$ ::= `Terminal`

Various special symbols are used on the right-hand side of a production rule. These symbols are explained in the following.

9.1.3.1 Parentheses

Ordinary parentheses are used around productions for metasyntactic grouping.

9.1.3.2 Alternatives

Alternative productions are separated by the metasyntactic infix operator $|$.

9.1.3.3 Optional Productions

Optional productions (that is, productions which may occur either zero or one time) are identified by the metasyntactic postfix operator $^?$.

9.1.3.4 Repeated Productions

Productions which may be repeated are identified by one the metasyntactic postfix operators * and $^+$. The former indicates that the production may be repeated any number of times, while the latter indicates that at least one occurrence is required.

9.1.4 Scope of Grammar Rules

The grammar rules given in this reference manual are to be taken as global definitions for the entire document.

9.2 Semantics Specifications

Throughout the AgentC documentation a special notation is used to describe how the ACC translates AgentC code into Java ditto. The notation not only documents how the ACC handles each language construct, but also defines the precise semantics of the AgentC code *in terms of the corresponding* Java *code.*

The notation uses a special *semantic function* χ; the argument to this function is shown as an AgentC language construct where some of the tokens are replaced by syntactic variables, which will be Greek lowercase letters; the corresponding Java code is then shown, using the same syntactic variables. It should be clear from the context (i.e., from the grammar rules for the language construct specified) what the syntactic variables should be taken to mean.

An auxiliary semantic function Δ is defined in section 15.4. The special notational suffix $^{-1}$ is defined in section 15.3.1.

An example semantic specification is found below:

$$\chi(\epsilon_1 \texttt{ == } \epsilon_2)$$

$$\equiv$$

$$\texttt{isEqual}(\chi(\epsilon_1), \chi(\epsilon_2))$$

The specification shows that the compiler for an AgentC comparison condition$_{15.3.1.1}$ in case of an `==` operator produces the Java code to invoke the `isEqual`$_{5.1}$ method on the compiled code of the two operands.

Chapter 10

Lexical Structure

10.1 Input Alphabet

The input alphabet of an AgentC program consists of the first 256 characters of the Unicode character set. However, only characters among the first 128 of these characters, corresponding to the standard ASCII character set, are necessary to write an AgentC program.

10.2 Lexical Translations

The lexical analyser (*lexer*) automatically decodes Unicode escape sequences (cf. [17] §3.3) in the input; in this way the full Unicode character set can be produced. Since only the ASCII character set is necessary to write the program, Unicode escapes are provided with the sole purpose of allowing arbitrary string literals$_{10.9.3}$ to be produced.

10.3 White Space

The characters whose only function are to separate *tokens* are called *white space characters*. The lexer treats the first 33 characters of the input alphabet as white space.

10.4 Comments

Comments are sequences of input which the lexer treats as white space. AgentC shares the lexical syntax of comments with Java; cf. [17] §3.7.

10.5 Identifiers

The first character of *Identifiers* in AgentC must belong to a restricted class:

$$\begin{array}{lll}
\langle\text{Identifier}\rangle & ::= & \langle\text{IdentifierStart}\rangle\ \big(\langle\text{Letter}\rangle \mid \langle\text{Digit}\rangle\big)^{*} \\
\langle\text{IdentifierStart}\rangle & ::= & \texttt{a ... z} \mid \texttt{\$}\ \langle\text{Letter}\rangle \\
\langle\text{Letter}\rangle & ::= & \texttt{A ... Z} \mid \texttt{a ... z} \mid \texttt{\_} \\
\langle\text{Digit}\rangle & ::= & \texttt{0} \mid \langle\text{NonZeroDigit}\rangle \\
\langle\text{NonZeroDigit}\rangle & ::= & \texttt{1 ... 9}
\end{array}$$

The main reason for this is that the lexer, which has been generated by JavaCC [21], does not allow keywords to be recognised as a subset of the legal identifiers — the lexical syntax must be unique.

Since all AgentC keywords consist of uppercase letters, the first letter of identifiers must then be a lowercase letter in order to differentiate it from a keyword token. The `$` character serves as an escape character (it will itself not be part of the identifier), such that it is possible to create identifiers whose first character is an uppercase letter.

10.6 Variables

Variables in AgentC use a special syntax distinct from the syntax of identifiers:

$$\langle\text{Variable}\rangle \quad ::= \quad \texttt{?}\ \langle\text{Letter}\rangle\ \big(\langle\text{Letter}\rangle \mid \langle\text{Digit}\rangle\big)^{*} \mid \texttt{\_}$$

Apart from the reasons given in section 10.5, the syntax has been chosen to resemble the variable syntax originally used in Agent-0 [30].

10.6.1 Wildcards

The variable whose syntax is _ has a special significance. It is denoted the *wildcard variable*.

10.7 Attitude Tokens

The AgentC language uses a set of symbolic names, called *attitude tokens*, in the syntax of sentences$_{14.7}$. These attitude tokens as well as their assigned values, called *attitudes*, are defined by the AgentC programmer. However, all attitude tokens share a common lexical syntax:

$$\langle\text{AttitudeToken}\rangle \quad ::= \quad \texttt{\#}\ \langle\text{Letter}\rangle\ \big(\langle\text{Letter}\rangle \ \big|\ \langle\text{Digit}\rangle\big)^*$$

10.8 Keywords

The *keywords* of AgentC are used to distinguish different syntactical constructs, to be defined later. The full list of AgentC keywords is the following:

```
ADOPT
AS
ATTITUDES
CALL
DEFS
DO
DROP
ELSE
ELSIF
FACTS
IF
LET
LOCKED
NOTHING
PROCEDURE
Q
RETURN
SAY
WHEN
XEQ
```

10.9 Literals

The simple values of the AgentC language are all produced by means of *literals*. Three different kinds of literals are used:

$$\langle\text{Literal}\rangle \quad ::= \quad \langle\text{IntegerLiteral}\rangle \ \big|\ \langle\text{DoubleLiteral}\rangle \ \big|\ \langle\text{StringLiteral}\rangle$$

10.9.1 Integer Literals

Integer literals produce integral numeric values. The syntax of integer literals is as decimal integer literals in Java ([17] §3.10.1), except that the negation sign is considered part of the literal in AgentC, and that no type suffix character is used:

$$\langle\text{IntegerLiteral}\rangle \quad ::= \quad \texttt{0} \;\big|\; \texttt{-}^{?}\ \langle\text{NonZeroDigit}\rangle\ \langle\text{Digit}\rangle^{*}$$

10.9.2 Double Literals

Double literals produce floating-point values. The syntax of double literals is similar to the syntax of floating-point literals in Java ([17] §3.10.2), except that no type suffix character is used:

$$\begin{array}{lcl}
\langle\text{DoubleLiteral}\rangle & ::= & \langle\text{Digit}\rangle^{+}\ \texttt{.}\ \langle\text{Digit}\rangle^{*}\ \langle\text{Exponent}\rangle^{?} \\
 & & \big|\ \texttt{.}\ \langle\text{Digit}\rangle^{+}\ \langle\text{Exponent}\rangle^{?} \\
 & & \big|\ \langle\text{Digit}\rangle^{+}\ \langle\text{Exponent}\rangle \\
\langle\text{Exponent}\rangle & ::= & \big(\texttt{e} \;\big|\; \texttt{E}\big)\ \big(\texttt{+} \;\big|\; \texttt{-}\big)^{?}\ \langle\text{Digit}\rangle^{+}
\end{array}$$

10.9.3 String Literals

String literals produce string values. The syntax of string literals in AgentC is similar to the syntax of string literals in Java, except that the input alphabet, as already defined$_{10.1}$, is different. Please refer to [17] §3.10.5–6 for a full definition of the string literal syntax.

Chapter 11

Types & Values

Internally, all AgentC values are represented as `Object` instances, and the compiled code mainly uses variables typed as `Object` to represent them. Only a few different kinds of values can be directly produced by means of AgentC code, however, and these are explained in the following sections.

Although only a few different kinds of values are inherent to the AgentC language, no type system per se is offered by the ACC: *all variables are treated as having the same type.*

The main reason that AgentC has such a simple type system is that only little is to be gained by providing a strict type system as e.g. Java does: the ACC must internally use a compile-time type of `Object` for all values, because they are represented by classes which do not share a common interface and whose only possible common superclass is `java.lang.Object`. Although the specific type of a simple value could be inferred from the way it is produced (i.e. from the literal), the type of a value obtained by a query$_{14.6}$ cannot be determined during compilation, since the common `Object` type must be used at that time. Instead of introducing type names and requiring explicit casts, as Java does (c.f. [17] §15.16), it seems a better choice to avoid using different types and to rely on Java's runtime type check to detect type errors when they occur (this will happen anyway). By doing so, a cleaner language and a simpler compiler implementation is gained, and future extensions to the language are made easier. The only price of this choice is that it is more difficult to detect type inconsistencies during compilation — it is impossible to determine *all* type inconsistencies at compile time in Java.

11.1 Simple Values

The *simple values* of AgentC are the values which are produced by means of a literal$_{10.9}$.

11.1.1 Integer Values

Integer values result from the an integer literal$_{10.9.1}$. Internally in the program, such a value is represented by an instance of `java.lang.Integer`.

11.1.2 Double Values

Double values are the result of a double literal$_{10.9.2}$. Internally in the program, such a value is represented by an instance of `java.lang.Double`.

11.1.3 String Values

String values result from string literals$_{10.9.3}$. Internally in the program, such a value is represented by an instance of `java.lang.String`.

11.2 Logical Values

Logical values are created by means of a sentence$_{14.7}$. Due to the nature of the AgentC language, a *very* simple set of sentences is allowed: a sentence is simply a DATALOG fact (c.f. [29] §6.2.4), combined with an integer attitude.

Normally, logical values are stored in a compact form in the ACME's knowledge base, which only allows ground facts to be stored. Sentences can also be extracted from the knowledge base or received in a message, in addition to being created directly by a sentence. In such a case the logical value will be represented by an instance of `dk.dtu.imm.cse.agent.act.afc.Fact`.

Chapter 12

Scope Rules

Every time a variable occurs in the AgentC program, it has one of two different meanings: either it is a *variable def* or a *variable use*. In a variable use the variable serves as a reference to a value which was previously evaluated and bound to the symbolic variable in a preceding variable def. A variable def is said to *bind* a value, which can then be referenced at a later point by a variable use.

The way in which the ACC classifies variable occurrences into variable defs and variable uses is determined by the lexical *scope* of these variable occurrences. In general, each block$_{15}$ constitutes a new scope, which initially contains the bindings of the enclosing scope (if any). At the end of the block the enclosing scope will be restored. Certain language constructs may also produce a new scope, or temporarily enforce the scope to have a certain type (typically read-only]); if so, the detailed documentation of these constructs will mention the fact.

The variable classification scheme is subject to a set of general rules, which depend on the kind of scope in which the variable occurs: a scope may be either read-only, write-only or read-write. In case a language construct enforces a specific kind of scope rather than using the enclosing scope (which it normally will), the detailed language documentation will mention this fact.

The scope rules rules employed by the ACC are listed below. Each rule is checked in the order listed here, and the first matching rule will determine how the variable is classified.

- Occurrences of wildcard variables$_{10.6.1}$ will always be treated as variable defs. Thus, no wildcard variable can ever refer to the same value. It is a semantic error to use a wildcard variable in a read-only scope.
- In a write-only scope, any variable occurrence will be a variable def.
- In a read-only scope, any variable occurrence will be a variable use. It is a semantic error not to let a variable occurring in a read-only scope be preceded by a corresponding variable def in the current or an enclosing scope.
- In a read-write scope, two possibilities exist:
  - If the variable has not been preceded by a variable def in the current or enclosing scope, then the occurrence will be a variable def.
  - Otherwise, the occurrence will be a variable use referring to the preceding variable def which is closest to the occurrence.

Chapter 13

Program Structure

An AgentC program consists of a number of *compilation modules*, which are to be defined in this and the following sections. The shorter term *module* will be used instead of compilation module in the rest of this part of the document.

The order in which the modules are specified is not significant, and except for procedures, the order in which their contents are specified is not significant either. The full program is combined from a set of modules, which can be placed in any number of files. In this way the programmer is free to structure the program as (s)he pleases, and to reuse common definitions.

The syntax of an AgentC program is:

$$\langle \text{Program} \rangle \quad ::= \quad \langle \text{CompilationModule} \rangle^*$$

A module is defined to be

$$\begin{array}{lll} \langle \text{CompilationModule} \rangle & ::= & \langle \text{Definitions} \rangle \mid \langle \text{Attitudes} \rangle \mid \\ & & \langle \text{Facts} \rangle \mid \langle \text{Procedure} \rangle \end{array}$$

13.1 Symbol Definitions

The *symbol definition* module allows a set of constant values to be defined under symbolic names. These names can then be used in the program as symbolic reference expressions$_{14.3}$.

In addition to making it easier to reuse and maintain common code, the symbol definition module also helps to optimise the resulting code, since the values for the symbolic reference expressions are instantiaed only once[1].

Syntax:

$$\begin{array}{lcl} \langle\text{Definitions}\rangle & ::= & \texttt{DEFS}\ \{\ \langle\text{Definition}\rangle^*\ \} \\ \langle\text{Definition}\rangle & ::= & \langle\text{Identifier}\rangle\ \texttt{=}\ \langle\text{Literal}\rangle\ \texttt{;} \end{array}$$

Notes:

It is a semantic error to use the same identifier twice unless the right-hand value is identical in both definitions. Assuming that `x = 2.0;` was defined previously, then `x = 2e0` would be legal, because `2.0` and `2e0` represent the same value; similarly, `x = 2;` would be illegal, since the integer literal `2` does not represent the same value as the double literal `2.0`.

Semantics:

For each definition ι `=` λ the ACC generates a field in the output ACME. Assuming that the Java type of the literal is τ, the generated code is specified by

$$\chi(\iota = \lambda)$$

$$\equiv$$

$$\underline{\textbf{public}}\ \underline{\textbf{static}}\ \underline{\textbf{final}}\ \tau\ \texttt{C\_}\iota = \chi(\lambda)\ ;$$

13.2 Attitude Declarations

In contrast to most other APLs, AgentC does not specify a fixed set of mental attitudes to be used — the programmer is free to choose the set of attitudes (s)he finds to be appropriate for the task. The choice of attitudes is specified by means of the *attitude declaration* module.

[1] as shown in section 14.1, every use of a numeric literal results in a Java instance creation expression. This is not the case for a symbolic reference expression$_{14.3}$

Syntax:

```
⟨Attitudes⟩       ::=  ⟨AttitudeDecl⟩*
⟨AttitudeDecl⟩    ::=  ⟨AttitudeToken⟩ = ⟨IntegerLiteral⟩ ;
```

Notes:

It is a semantic error to specify a negative number, or to assign different attitudes to the same attitude token. Otherwise, it is legal to repeat definitions, or to declare the same attitude under different names.

It is strongly recommended to use the same attitude declarations for all ACMEs in a given application, since the compiled program does not use the symbolical names, but rather the declared integer attitudes. It is the responsibility of the programmer to ensure that sentences communicated between different ACMEs are handled appropriately — and this is most easily ensured by using a consistent set of attitude declarations.

It is also strongly recommended to use reasonably small values in the attitude declarations, since the initial size of the resulting knowledge base will be directly proportional to the largest value specified.

Semantics:

The declarations given in the attitude declaration module are indirectly used to translate a attitude$_{10.7}$ into an integer value. However, given a attitude declaration module whose largest value is ω, the compiler produces the following method in the ACME:

```
public int getMaxAttitude() {
    return $\omega$;
}
```

This method is used in construction of the knowledge base which, as already mentioned, has an initial size proportional to ω[2]

[2] The knowledge base internally uses an array of containers to hold its contents; a given fact is then placed in the container indexed by its attitude. The value returned by `getMaxAttitude` then determines the length of this array

13.3 Initial Facts

The *initial facts* module provides an easy way to specify the initial contents of the knowledge base.

Syntax:

$$\begin{array}{lll} \langle\text{Facts}\rangle & ::= & \langle\text{InitialFact}\rangle^* \\ \langle\text{InitialFact}\rangle & ::= & \langle\text{Fact}\rangle\ \ ; \end{array}$$

Notes:

Although the syntax of a ⟨Fact⟩ allows variables, it is a semantic error to use a variable inside the initial facts.

The compiler does not check for duplicate facts in the specification — the result of a duplicate will be a less than optimal program, but the contents of the knowledge base will not be affected by a duplicate addition.

Semantics:

For each initial fact ϕ, the compiler adds

```
kb.add(χ(φ));
```

to the body of the method

```
protected void initKnowledgeBase(KnowledgeBase kb) {
}
```

which is invoked by the ACME when it is initialised.

13.4 Procedures

Procedures are the main modules of an AgentC program, as they result in executable code whereas the other kinds of modules produce or configure the data of the program.

Since the intra-agent architecture is not restricted by AgentC, there is no requirement regarding which procedures to include in the program, and what their names should be — it is up to the programmer to determine the number and names of the procedures to be included, and how to call these from outside the ACME.

Syntax:

$$
\begin{array}{lcl}
\langle\text{Procedure}\rangle & ::= & \texttt{PROCEDURE}\ \langle\text{Identifier}\rangle\ \langle\text{ParameterList}\rangle \\
 & & \big(\texttt{;}\ \mid\ \langle\text{ProcedureBody}\rangle\big) \\
\langle\text{ParameterList}\rangle & ::= & \texttt{(}\ \big(\langle\text{Variable}\rangle\ \big(\texttt{,}\ \langle\text{Variable}\rangle\big)^{*}\big)^{?}\ \texttt{)} \\
\langle\text{ProcedureBody}\rangle & ::= & \texttt{\{}\ \langle\text{Rule}\rangle^{*}\ \texttt{\}}
\end{array}
$$

Notes:

The combination of a procedure name and a parameter list is known as a *procedure signature*. Different procedures may use the same identifier, as long as the procedure signatures are different.

- If no body is given, then the only result of the declaration — which in this case is known as a *procedure requirement declaration* — is to ensure that the given procedure signature is usable in the rest of the program.
  A procedure requirement declaration can legally be specified any number of times, and only the number of variables is significant. However, it is a semantic error to omit specifying a corresponding procedure elsewhere in the program.
- Otherwise, the procedure whose behaviour is specified in the code of the body will be added to the program. It is a semantic error to declare a procedure body more than once, even if the declarations are identical; it is furthermore a semantic error to use a wildcard variable as part of the procedure signature, or to use the same variable more than once.
  The procedure body constitutes a new read-write scope, which initially contains the variables which were used in the procedure signature accompanying the procedure body.

Semantics:

For each procedure having a body,

$$\chi(\underline{\textbf{PROCEDURE}}\ \pi(\xi_0,\ldots,\xi_n)\ \texttt{\{}\rho_0\ \ldots\ \rho_m\texttt{\}})$$

$\equiv$

```
public Object $\pi$(Map msg, Object $\chi(\xi_0)$[3], ..., Object $\chi(\xi_n)$) {
    $\chi(\rho_0)$
    ...
    $\chi(\rho_m)$
}
```

13.4.1 Calling Procedures

The methods resulting from the compilation of a procedure can be used in two ways: either they are called directly from somewhere (normally the main execution loop) of the intra-agent architecture, or from within a procedure call$_{14.5}$ in the generated code.

For each invocation of the method, the caller must supply a value to the parameter `Map msg`, which is the next message to be handled. When invoking the method, the intra-agent architecture has full freedom to choose the value of this message; furthermore, the intra-agent architecture has full freedom to choose when, and how often to invoke the method — and to choose which methods to invoke.

In most cases, the main execution loop would probably look like this:

1. Wait until a message is available, or a timeout occurs.
2. (a) If the operation timed out, then supply `null` as a parameter.
   (b) Otherwise, supply the received message as a parameter.
3. Repeat.

13.4.2 Temporary Variables

The compiled code of many rules depend on one or more *temporary variables*. A temporary variable is a Java variable which the generated code uses to store an intermediate result. The ACC declares the necessary temporary variables at the beginning of a procedure; because of the way the ACC translates code, at most one temporary variable of each kind will be necessary, regardless of how the translated code is composed.

The possible temporary variables, in conjunction with their Java initialiser, are listed here:

[3] Here, $\chi(\xi)$ denotes the meaning defined for a variable use (ξ_v) in section 14.2, although the variables are technically variable defs.

```
BitSet bitSet = new BitSet();
KnowledgeBase kBase = getKnowledgeBase();
Object tempObj;
Map tempMap;
Fact tempFact;
```

Chapter 14

Values and Expressions

An AgentC *value* is either an *expression* or a sentence:

$\langle\text{Value}\rangle \quad ::= \quad \langle\text{Expression}\rangle \mid \langle\text{Sentence}\rangle$

where

$$\begin{aligned}\langle\text{Expression}\rangle \quad ::= \quad & \langle\text{Literal}\rangle \mid \langle\text{Variable}\rangle \mid \langle\text{SymbolReference}\rangle \mid \\ & \langle\text{SelfReference}\rangle \mid \langle\text{ProcedureCall}\rangle\end{aligned}$$

14.1 Literal Expressions

Any literal$_{10.9}$ constitutes a *literal expression*.

Semantics:

For each integer literal ι,

$\chi(\iota)$

$\equiv$

new `Integer`(ι)

For each double literal δ,

$\chi(\delta)$

$\equiv$

new `Double`(δ)

For each string literal σ,

$\chi(\sigma)$

$\equiv$

σ

14.2 Variable Expressions

Every occurence of a variable is a *variable expression*.

Notes:

As already mentioned in chapter 11, it is the responsibility of the programmer to ensure that the value stored in a variable has a type which is appropriate at the point of the variable expression.

Semantics:

During the semantic check performed before compilation, every variable def is assigned a unique ID, and for every variable use is is determined to which variable def — and hence to which unique ID — the variable use belongs. Given a variable use ξ_v referring to a variable def having ID ι, the corresponding Java code is an identifier whose name will be

$\chi(\xi_v)$

$\equiv$

$\mathtt{v}\iota\mathtt{\_}\xi_v$

The compilation of variable defs will result in separate code which will be described in the documentation for the language constructs allowing them. However, for the sake of completeness in the documentation, the default case for a variable def ξ_δ is defined as

$\chi(\xi_\delta)$

$\equiv$

<u>null</u>

14.3 Symbolic References

A *symbolic reference expression* is a reference to one of the global symbol definitions$_{13.1}$.

Syntax:

$\langle$SymbolReference$\rangle$ ::= $\langle$Identifier$\rangle$

Notes:

It is a semantic error to use a symbolic reference expression which has not been declared in at least one symbol definition module.

Semantics:

Symbolic reference expressions compile into a Java identifier which refers to the field produced by the symbol definition (c.f. section 13.1):

$\chi(\iota)$

$\equiv$

`C_`ι

14.4 The Self Reference

Each ACME is constructed using a unique `Object` instance to identify the ACME. The value of this ID can be referenced from within the AgentC program using the *self reference expression*.

Syntax:

$\langle$SelfReference$\rangle$ ::= `SELF`

Semantics:

$$\chi(\texttt{SELF})$$

$\equiv$

$$\texttt{getId}()$$

14.5 Procedure Calls

A *procedure call* allows the code of one procedure to call another procedure; the expression evaluates to the value returned$_{15.7}$ by the called procedure. The message parameter of the calling procedure will be transferred as the message parameter of the called procedure.

Syntax:

$\langle$ProcedureCall$\rangle$ ::= `CALL` $\langle$identifier$\rangle$ $\langle$TermList$\rangle$

Notes:

The $\langle$TermList$\rangle$ has a read-only scope.

It is a semantic error to use a procedure signature which is not defined elsewhere in the program.

Semantics:

$$\chi(\underline{\textbf{CALL}}\ \pi(\tau_0, \ldots, \tau_n))$$

$\equiv$

$$\pi(\texttt{msg},\ \chi(\tau_0)\,,\ \ldots\ ,\ \chi(\tau_n)\,)$$

14.6 Queries

A *query* allows the AgentC program access or evaluate values which cannot otherwise be produced in AgentC code. The request for information is serviced by the investigator of the ACME, which should have been specially written in order to support the possible requests of the program.

Syntax:

$$\begin{array}{lll}\langle\text{Query}\rangle & ::= & \texttt{Q}\ \langle\text{Identifier}\rangle\ \langle\text{TermList}\rangle \\ \langle\text{TermList}\rangle & ::= & \texttt{(}\ \big(\langle\text{Expression}\rangle\ \big(\texttt{,}\ \langle\text{Expression}\rangle\big)^{*}\big)^{?}\ \texttt{)}\end{array}$$

Notes:

The query itself has a read-only scope.

Semantics:

Given a $\langle\text{TermList}\rangle\ \tau = (\tau_0, \tau_1, \ldots, \tau_n)$,

$$\chi(\underline{\mathbf{Q}}\ \iota\ \tau)$$

$\equiv$

$$\texttt{query}(\iota,\ \chi(\tau))$$

where

$$\chi(\tau)$$

$\equiv$

$$\underline{\textbf{new}}\ \texttt{Object}[]\ \{\ \chi(\tau_0),\ \chi(\tau_1),\ \ldots,\ \chi(\tau_n)\ \ \}$$

14.7 Sentences

Sentences in AgentC are simply facts:

$$\langle\text{Sentence}\rangle\quad ::=\quad \langle\text{Fact}\rangle$$

The reason why this document distinguished between sentences and facts may be a bit obscure; however, the word *sentence* is used with reference to a general class of language constructs which at the moment only count facts, but which may be extended in future versions of the language. *Fact*, however, is used with reference to the specific syntactical construct defined in section 14.7.1.

14.7.1 Facts

As mentioned in section 11.2, a *fact* in AgentC corresponds to a DATALOG fact combined with an attitude.

Syntax:

$$\langle \text{Fact} \rangle \quad ::= \quad \langle \text{AttitudeToken} \rangle \ \langle \text{Identifier} \rangle \ \langle \text{TermList} \rangle$$

Notes:

It is a semantic error to specify an attitude token for which no attitude value has been declared in an attitude declaration$_{13.2}$.

Semantics:

$$\chi(\alpha \ \iota \ \tau)$$

$$\equiv$$

$$\chi(\alpha) \, , \, \iota \, , \, \chi(\tau)$$

Here, $\chi(\alpha)$ simply results in the attitude declared for the attitude token α.

Note that the compiled code does not in itself constitute a valid Java expression. The reason is that the ACC tries to avoid creating a new `Fact` instance, if possible; the specific occurrence of the fact will then determine how the code above will be used.

Chapter 15

Rules and Statements

Rules are the building blocks of a procedure. A rule is defined as

$$\langle\text{Rule}\rangle \quad ::= \quad \langle\text{Statement}\rangle \;\big|\; \langle\text{MessageRule}\rangle$$

where

$$\begin{array}{lll}\langle\text{Statement}\rangle & ::= & \langle\text{Action}\rangle \;\big|\; \langle\text{Assignment}\rangle \;\big|\; \langle\text{IfStatement}\rangle \;\big| \\ & & \langle\text{LetStatement}\rangle \;\big|\; \langle\text{LockedStatement}\rangle \;\big| \\ & & \langle\text{MentalUpdate}\rangle \;\big|\; \langle\text{ReturnStatement}\rangle \;\big| \\ & & \langle\text{SayStatement}\rangle\end{array}$$

Each kind of rule is explained in the following sections. Common to the syntax of some of these rules is the notion of a *block*, which simply is a sequence of statements enclosed in braces:

$$\langle\text{Block}\rangle \quad ::= \quad \{ \;\; \langle\text{Statement}\rangle^* \;\; \}$$

For a block containing statements $\sigma_0 \ \dots \ \sigma_n$,

$$\chi(\{\ \sigma_0 \ \dots \ \sigma_n \ \})$$

$\equiv$

$$\begin{array}{l}\chi(\sigma_0) \\ \dots \\ \chi(\sigma_n)\end{array}$$

15.1 Action Statements

The `DO`-*statement* and `XEQ`-*statement* are collectively known as *action statements*. Common to both statements is that they delegate to the ACME's actuator to execute an operation identified by a string, and that the success of this operation can be queried by using the statement as a condition$_{15.3.1}$.

Syntax:

$$\begin{array}{lcl} \langle\text{ActionStatement}\rangle & ::= & \langle\text{DoStatement}\rangle \mid \langle\text{XeqStatement}\rangle \\ \langle\text{DoStatement}\rangle & ::= & \texttt{DO}\ \langle\text{Identifier}\rangle\ \langle\text{TermList}\rangle\ \texttt{;} \\ \langle\text{XeqStatement}\rangle & ::= & \texttt{XEQ}\ \langle\text{Identifier}\rangle\ \langle\text{TermList}\rangle\ \texttt{;} \end{array}$$

Notes:

The whole action statement constitutes a read-only scope.

The only difference between the two kinds of statements is that the `DO`-statement updates the knowledge base when it has been executed; assume that `#DID` has been declared as the achievement attitude$_{15.1.1}$. Then

$$\underline{\textbf{DO}}\ \alpha(\tau_0, \tau_1, \ldots, \tau_n)$$

will have the same effect as

```
IF (XEQ α(τ_0, τ_1, ..., τ_n)) {
    ADOPT #DID α(τ_0, τ_1, ..., τ_n);
}
```

Semantics:

$$\chi(\underline{\textbf{XEQ}}\ \alpha\ \tau)$$

$\equiv$

$$\texttt{xeqAction}(\alpha,\ \chi(\tau))$$

$$\chi(\underline{\textbf{DO}}\ \alpha\ \tau)$$

$\equiv$

$$\texttt{doAction}(\alpha,\ \chi(\tau))$$

15.1.1 The Achievement Attitude

The *achievement attitude* is a special attitude which is used in the `DO`-statement, as already shown in the preceding compiler specification. The value of the achievement attitude is determined by the return value of the ACME's `getAchievementId` method, which by default is `0`.

It is the responsibility of the ACME programmer to ensure that the achievement attitude does not interfere with any other attitude in the program. This *could* be done by overriding the `getAchievementId` method, but for the sake of interoperability, it is recommended to reserve attitude 0 for this purpose, regardless of whether the `DO`-statement is used or not, and to avoid declaring attitudes having the value 0 unless they will be used exactly for this purpose. The price for such a reservation will only be a small increase in the size of the knowledge base[1].

15.2 Assignment Statements

An *asignment statement* updates the value of a previously bound variable to a new value.

Syntax:

$$\langle\text{Assigmnent}\rangle \quad ::= \quad \langle\text{Variable}\rangle \; \texttt{=} \; \langle\text{Expression}\rangle \;\; \texttt{;}$$

Notes:

The asignment statement has a read-only scope.

It is a semantic error to assign to an undefined variable, a wildcard variable or to a variable which is not assignable$_{15.4}$.

Semantics:

$$\chi(\xi \; = \epsilon)$$

$$\equiv$$

$$\chi(\xi) \; = \chi(\epsilon) \; ;$$

[1] The extra memory amounts to an empty `Map` instance and a reference to it

15.3 The IF Statement

The IF-*statement* is by far the most complex language construct in AgentC, although its syntax is deceptively simple:

Syntax:

$$
\begin{array}{lll}
\langle\text{IfStatement}\rangle & ::= & \texttt{IF}\ \langle\text{Guard}\rangle\ \langle\text{Block}\rangle \\
 & & \big(\texttt{ELSIF}\ \langle\text{Guard}\rangle\ \langle\text{Block}\rangle\big)^{*} \\
 & & \big(\texttt{ELSE}\ \langle\text{Block}\rangle\big)^{?} \\
\langle\text{Guard}\rangle & ::= & \texttt{(}\ \big(\langle\text{Condition}\rangle\ \big(\texttt{,}\ \langle\text{Condition}\rangle\big)^{*}\big)^{?}\ \texttt{)}
\end{array}
$$

Notes:

A block contained in an IF-statement are called *branches*.

Loosely defined, the IF-statement executes as follows:

- For each branch
  - ○ For each guard condition
    - ∗ If the condition is not fulfilled, then continue to the next branch.
    - ∗ Otherwise, continue to the next condition
  - ○ Execute the branch *once for each possible combination of values which fulfil the guard condition(s)*, then exit the loop.

Hence, at most once branch can be executed, but it can execute any number of times, depending on the guard conditions.

15.3.1 Conditions

A *condition* is a constraint on some expression values, or on the state of the knowledge base. A *simple condition* is a condition which has only two possible outcomes: either it is fulfilled, or it is not. A *complex condition*, on the other hand, can be fulfilled in more than one way.

All simple conditions have an *inverse condition*, which is fulfilled exactly when the condition itself is not. The compiler specifications found in the following sections all mention how the inverse condition is produced. The suffix notation $^{-1}$ is used on syntactic variables to specify that the inverse condition is used.

The possible conditions are:

$$\langle\text{Condition}\rangle ::= \langle\text{Comparison}\rangle \mid \langle\text{NegatedCondition}\rangle \mid \langle\text{MentalCondition}\rangle \mid \langle\text{Action}\rangle \mid \texttt{(}\ \langle\text{Condition}\rangle\ \texttt{)}$$

15.3.1.1 Comparison Conditions

Comparison conditions are simple conditions which are fulfilled only if the two operand expressions fulfil the given relation.

Syntax:

$$\begin{aligned}\langle\text{Comparison}\rangle &::= \langle\text{Expression}\rangle\ \langle\text{Relation}\rangle\ \langle\text{Expression}\rangle\\ \langle\text{Relation}\rangle &::= \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{>}\end{aligned}$$

Notes:

The entire comparison condition has a read-only scope.

It is the responsibility of the programmer to ensure that the types of the two operand expression are compatible — otherwise a `ClassCastException` will result when the condition is evaluated.

Semantics:

$$\chi(\epsilon_1\ == \epsilon_2)$$

$$\equiv$$

$$\texttt{isequal}(\chi(\epsilon_1), \chi(\epsilon_2))$$

$$\chi(\epsilon_1\ != \epsilon_2)$$

$$\equiv$$

$$\texttt{!isequal}(\chi(\epsilon_1), \chi(\epsilon_2))$$

$$\chi(\epsilon_1\ \rho\ \epsilon_2)$$

$$\equiv$$

$$(((\texttt{Comparable})\chi(\epsilon_1)).\texttt{compareTo}(\chi(\epsilon_2))\ \rho\ 0$$

The inverse condition is handled as shown above, with the exception that the symbols `==` and `!=` should be exchanged, and that the second occurrence of ρ should be replaced by ρ^{-1}.

15.3.1.2 Negated Conditions

A *negated condition* is a simple condition which behaves as the inverse condition of its operand.

Syntax:

⟨NegatedCondition⟩ ::= `!` ⟨Condition⟩

Notes:

The negated condition has a read-only scope

Semantics:

$$\chi(!\ \zeta)$$

$\equiv$

$$\chi(\zeta^{-1})$$

15.3.1.3 Action Statements as Conditions

An action statement$_{15.1}$ can be used as a simple condition. The condition is fulfilled only if the action could be successfully executed. Hence, an attempt to execute the action will occur when the condition is evaluated, regardless of whether the corresponding branch will be executed.

Notes:

The action statement has a read-only scope

Semantics:

As defined in section 15.1. The inverse condition is obtained by prefixing a `!` to the compiled code.

15.3.1.4 Mental Conditions

The fulfilment of *mental conditions* depends on the state of the knowledge base. A mental condition is a simple condition iff it is a ground fact, and a complex condition otherwise.

Syntax:

$$\langle\text{MentalCondition}\rangle \quad ::= \quad \langle\text{Fact}\rangle \; \big(\texttt{AS} \; \langle\text{Variable}\rangle\big)^{?}$$

Notes:

The optional variable following the fact is known as an *alias*; it will be updated to contain the fact on its left every time the mental condition is found to be fulfilled, and its scope extends from the location immediately to the right of the alias and ends with the branch in which it occurs.

In the simple case, the mental condition is fulfilled iff the given ground fact exists in the knowledge base.

In the complex case, the mental condition is fulfilled once for every ground fact in the knowledge base which matches the given pattern. Let #B be an attitude token; if the knowledge base contains the three facts

```
#B p(”a”, 0)
#B p(”b”, 0)
#B p(”c”, 1)
```

then the mental condition

```
#B p(?X, 0)
```

would be fulfilled twice, and the variable ?X would be bound once to the value "a" and once to the value "b". The order in which these bindings would occur is not generally known, but depends on the specific knowledge base[2].

[2] The specific knowledge base implementation in use by the ACME determines the order in which facts are retrieved, and hence how the bindings will occur. The knowledge base used by default — dk.dtu.imm.cse.agent.act.afc.AcmeKnowledgeBase — delivers matching facts in the iteration order of the LinkedHashSet internally used to store them. By default, therefore, facts will be returned in the order they were originally stored in the knowledge base.

Semantics:

For a ground fact ϕ used as a mental condition,

$\chi(\phi)$

$\equiv$

`kBase.contains`$(\chi(\phi))$

where the first ϕ refers to a *mental condition*, while the second ϕ refers to a *fact*, for which $\chi(\phi)$ has been defined in in section 14.7.1. The inverse condition will result in an additional `!` in front of the code.

The complex case is described in section 15.3.3.

15.3.2 About the Order of Conditions

Conceptually, the `IF`-statement resembles a DATALOG definite clause (c.f. [29] §6.2.2): the head of the clause may be viewed as the branch of the `IF`-statement, while the body of the clause corresponds to the list of conditions; in both cases, variables are implicitly universally quantified.

For a number of reasons, the order in which conditions are specified is significant, despite that the conditions implicitly are joined by a logical conjunction, which in itself is commutative:

1. The occurrence of variable defs may make it impossible to swap two conditions$_{15.3.2.1}$
2. Conditions may have side-effects$_{15.3.2.2}$
3. Efficiency of the resulting code depends on the order in which conditions occur$_{15.3.2.3}$

15.3.2.1 Scope Rules in Conditions

As the preceding descriptions of the various conditions have shown, only the mental condition allows variable defs, since it is the only kind of condition having a read-write scope; the occurence of a variable defs which is not an alias will make the mental condition a complex condition. Since the general scope rules$_{12}$ also apply to the `IF`-statement, a variable use cannot precede a variable def, because the scope of a variable def only extends forwards till the end of the branch.

For example, if ?X is not found in the enclosing scope, then

```
IF (Q q(?X) < 0, #B p(?X)) {
    ...
}
```

would be illegal, because the first occurrence of the variable ?X occurs in a read-only scope (in the query). On the other hand,

```
IF (#B p(?X), Q q(?X) < 0) {
    ...
}
```

is legal, since the unbound variable ?X first occurs within a read-write scope (in the mental condition), which affects the scope of the remaining branch, such that ?X is not unbound inside the read-only scope of the query.

15.3.2.2 Conditions with Side-effects

Generally speaking, many conditions may have side-effects, that is, their evaluation may affect the state of the program (ACME). For example, an action statement is used solely for its side-effect. In many cases procedure calls would be used for their side-effects rather than their result, and, depending on the specific investigator, even a query might have a side-effect.

Because the ACC cannot determine which conditions will have side-effects (let alone which of these side-effects are intended), the ACC does not try to optimise code by rearranging conditions — this is the job of the AgentC programmer$_{15.3.2.3}$.

15.3.2.3 Efficient Condition Evaluation

Let $c_0, \ldots, c_n$ be a series of conditions, and ϕ_i denote the number of times condition c_i is fulfilled. Furthermore, let ϵ_i denote the number of times condition c_i is evaluated. Then,

$$\epsilon_i = \prod_{j=0}^{i-1} \phi_j$$

In other words, every condition which is fulfilled n times will cause the following condition to be evaluated n times as many as the preceding condition (assuming that all conditions are fulfilled at least once).

In order to ensure that the whole IF-statement can be executed as efficiently, it is therefore wise to place conditions in increasing order of complexity, if possible, such that the combined number of condition evaluations is minimised, and such that more expensive condition evaluations[3] are performed after the less expensive conditions evaluations[4]. For example,

```
IF (#B p(?X, ?Y), Q q("") == 0) {
    ...
}
```

would evaluate `Q q("") == 0` three times (if it were fulfilled), assuming the same example knowledge base as shown in section 15.3.1.4. It would be much more efficient to use

```
IF (Q q("") == 0, #B p(?X, ?Y)) {
    ...
}
```

since `Q q("") == 0` would then only be evaluated once. Furthermore, in case it were not fulfilled, the evaluation of the complex condition would be avoided altogether.

15.3.3 Semantics of the IF Statement

In the general case, the IF-statement can result in quite complex code. In order to preserve readability, the semantics description is therefore given in parts rather than all at once.

Consider a general IF-statement

[3] The evaluation of a complex condition requires a knowledge base `match` operation$_{5.3.3.2}$, whereas a simple condition only requires a knowledge base `contains` operation$_{5.3.3.2}$.

[4] Generally, evaluation of a negated condition or a comparison condition has the same computational complexity as evaluation of the constituent condition(s). The computational complexity involved in evaluating an action statement depends on the actuator implementation, while the expense of a procedure call depends on the contents of the code for that procedure.

```
IF ($\gamma_{00}, \ldots, \gamma_{0\lambda_0}$) {
    $\beta_0$
}
ELSIF ($\gamma_{10}, \ldots, \gamma_{1\lambda_1}$) {
    $\beta_1$
}
...
ELSIF ($\gamma_{(n-1)0}, \ldots, \gamma_{(n-1)\lambda_{n-1}}$) {
    $\beta_{n-1}$
}
ELSE {
    $\beta_n$
}
```

Let $\Xi(i, j)$ be a semantic function which specifies the compilation of guard condition γ_{ij}, and let ν be denote an integer which can be used as a suffix in order to generate a new local variable[5]. Then the result would be

```
for (boolean matched$\nu$ = false; ; ) {
    do {
        $\Xi(0, 0)$
        matched$\nu$ = true;
        $\chi(\beta_0)$
    } while (false);
    if (matched$\nu$) break;
    do {
        $\Xi(1, 0)$
        matched$\nu$ = true;
        $\chi(\beta_1)$
    } while false;
    if (matched$\nu$) break;
    ...
    do {
        $\Xi(n-1, 0)$
        matched$\nu$ = true;
        $\chi(\beta_{n-1})$
    } while (false);
    if (matched$\nu$) break;
```

[5] The ACC uses the *indentation level* (i.e., the number of indentations prefixed on the Java statement in question) for this purpose.

```
        $\chi(\beta_n)$
        **break**;[6]

    }
```

When compiling a guard $\gamma_{i0}, \ldots, \gamma_{i\lambda_i}$, $\Xi(i,n)$ is recursively defined using 4 different cases.

- For $n > \lambda_i$, $\Xi(i,n)$ produces the empty string.
- If γ_{in} is a simple condition without an alias[7], then

```
        $\Xi(i,n)$

    $\equiv$

        if ($\chi(\gamma_{in}^{-1})$) break;
        $\Xi(i,n+1)$
```

- If γ_{in} is a simple condition having an alias α, then

```
        $\Xi(i,n)$

    $\equiv$

        tempFact = new Fact($\chi(\gamma_{in})$);
        if (!kBase.contains(tempFact)) break;
        $\Delta(\alpha$, tempFact)
        $\Xi(i,n+1)$
```

- If γ_{in} is a complex condition having variable defs $\xi_0, \ldots, \xi_{\mu_i}$ at indices $\delta_0, \ldots, \delta_{\mu_i}$, then

```
        $\Xi(i,n)$

    $\equiv$

        tempBitSet.clear();
        tempBitSet.set($\delta_0$);
        ...
        tempBitSet.set($\delta_{\mu_i}$);
        List match$\nu$ = kBase.match($\chi(\gamma_{in})$, tempBitSet);
```

[6] The break is not produced if $\chi(\beta_n)$ contains a RETURN-statement — otherwise the breakstatement would be unreachable.

[7] Even though it is legal to explicitly use a wildcard variable as an alias, it will technically not be an alias because it has no effect: the ACC represents the absence of an explicit alias in the same way as an explicit wildcard variable alias.

```
$\underline{\textbf{for}}$ ($\underline{\textbf{int}}$ $\texttt{i}\nu = 0$, $\texttt{max}\nu = \texttt{match}\nu\texttt{.size}()$; $\texttt{i}\nu < \texttt{max}\nu$; $\texttt{i}\nu$++) {
    $\texttt{tempFact} = (\texttt{Fact})\texttt{match}\nu\texttt{.get}(\texttt{i}\nu)$;
    $\Delta(\xi_0\,,\ \texttt{tempFact.getTerm}(\delta_0))$
    ...
    $\Delta(\xi_{\mu_i}\,,\ \texttt{tempFact.getTerm}(\delta_{\mu_i}))$
    $\Xi(i, n+1)$
}
```

In case the complex condition had an alias (other than a wildcard variable) α, then the line

$\Delta(\alpha\,,\ \texttt{tempFact})$

should be inserted in the above before $\Xi(i, n+1)$.

15.4 The LET Statement

The LET-*statement* introduces a new variable binding in the scope in which the statement occurs. The LET-statement is the only means of introducing an *assignable variable*, that is, a variable whose value can be modified explicitly in AgentC code$_{15.2}$.

Syntax:

⟨LetStatement⟩ ::= LET ⟨Variable⟩ = ⟨Expression⟩ ;

Notes:

The scope of the variable is a write-only scope; the scope of the right-hand expression is a read-only scope; after the statement, the enclosing scope will bind the variable to the expression — unless the variable is a wildcard variable, in which case the enclosing scope will not be modified and the evaluated value will be discarded.

Semantics:

$$\chi(\underline{\textbf{LET}}\ \_ = \epsilon)$$

$\equiv$

$\texttt{tempObj} = \chi(\epsilon);$

The reason that the ACC uses a temporary variable in this case is to avoid ending up producing a simple expression, which is not a legal Java statement; a more optimal solution would be to detect this case specially and to avoid producing code, since no visible effect would result from the simple expression evaluation.

Let ξ be a variable def, and let ι denote the corresponding ID. Then,

$\chi(\underline{\textbf{LET}}\ \xi = \epsilon)$

$\equiv$

$\texttt{Object v}\iota\texttt{\_}\xi = \chi(\epsilon);$

The above case generally demonstrates how the ACC produces code for a variable def, although the right-hand side may vary. For this purpose, the special notation

$\Delta(\xi, \epsilon)$

$\equiv$

$\texttt{Object v}\iota\texttt{\_}\xi = \epsilon;$

will be used instead.

15.5 The `LOCKED` Statement

The `LOCKED`-*statement* is a means to ensure that the contents of the knowledge base are consistent during execution of an entire block.

Syntax:

⟨LockedStatement⟩ ::= `LOCKED` ⟨Block⟩

Notes:

The block constitutes a new read-write scope.

Concurrent access to the knowledge base is disallowed for the entire execution of the block. It is legal to nest `LOCKED`-statements, although only the first of these will have any effect (except for a slight performance degradation).

Semantics:

$\chi($**LOCKED** $\beta)$

$\equiv$

```
synchronized (kBase.getLock()) {
    χ(β)
}
```

15.6 Mental Updates

Mental update statements are statements which direcly influence the contents of the knowledge base. Two kinds of mental update statements exist:

$\langle\text{MentalUpdate}\rangle \quad ::= \quad \langle\text{AdoptStatement}\rangle \mid \langle\text{DropStatement}\rangle$

15.6.1 The `ADOPT` statement

The `ADOPT`-*statement* places a fact in the knowledge base; if the sentence already existed in the knowledge base, the statement will not have any effect.

Syntax:

$\langle\text{AdoptStatement}\rangle \quad ::= \quad \texttt{ADOPT} \ \big(\langle\text{Fact}\rangle \mid \langle\text{Variable}\rangle\big) \ ;$

Notes:

The scope of the `ADOPT`-statement is a read-only scope.

In case the adopted value is a variable, it is the responsibility of the programmer to ensure that the variable contains a `Fact` instance.

Semantics:

For a variable ξ,

$\chi(\underline{\textbf{ADOPT}}\ \xi)$

$\equiv$

`kBase.add((Fact)`$\chi(\xi)$`);`

For a fact ϕ,

$\chi(\underline{\textbf{ADOPT}}\ \phi)$

$\equiv$

`kBase.add(`$\chi(\phi)$`);`

15.6.2 The `DROP` statement

The `DROP`-*statement* removes any number of facts from the knowledge base which match a given pattern.

Syntax:

$\langle\text{DropStatement}\rangle \quad ::= \quad$ `DROP` $\big(\langle\text{Fact}\rangle \mid \langle\text{Variable}\rangle\big)$ `;`

Notes:

If the right-hand side is a fact, it will belong to a read-write scope. A variable def will have the effect that any term will match the variable.

If the right-hand side is a variable, it will belong to a read-only scope. It is the responsibility of the programmer to ensure that the variable contains a `Fact` instance.

Semantics:

For a variable ξ,

$\chi(\underline{\textbf{DROP}}\ \xi)$

$\equiv$

```
kBase.remove((Fact)χ(ξ));
```

For a ground fact ϕ

$\chi(\underline{\textbf{DROP}}\ \phi)$

$\equiv$

```
kBase.remove(χ(φ));
```

For a fact ϕ having variable defs at indices $\delta_0, \ldots, \delta_n$,

$\chi(\underline{\textbf{DROP}}\ \phi)$

$\equiv$

```
tempBitSet.clear();
tempBitSet.set(δ_0);
...
tempBitSet.set(δ_n);
kBase.remove(χ(φ), tempBitSet);
```

15.7 The RETURN statement

The RETURN-*statement* determines which value should be returned from the method generated from a procedure, and thus the value resulting from a procedure call$_{14.5}$.

Syntax:

⟨ReturnStatement⟩ ::= RETURN ⟨Expression⟩$^?$;

Notes:

The expression enforces a read-only scope.

If no RETURN-statement is explicitly given in a procedure, the ACC will automatically insert RETURN; at the end of the procedure.

It is a semantic error to let the `RETURN`-statement — or a `LOCKED`-statement$_{15.5}$ whose block ends with a `RETURN`-statement — be succeeded by any rule inside the same block[8].

Semantics:

$$\chi(\underline{\textbf{RETURN}};)$$

$$\equiv$$

$$\underline{\textbf{return}}\ \underline{\textbf{null}};$$

$$\chi(\underline{\textbf{RETURN}}\ \epsilon;)$$

$$\equiv$$

$$\underline{\textbf{return}}\ \chi(\epsilon);$$

15.8 The `SAY` statement

The `SAY`-*statement* allows the ACME to send an arbitrary message. The message is sent using the ACME's messenger, which, depending on the implementation, determines how the addressee should be specified and how to transport the message to the addressee.

Syntax:

$$\begin{array}{lll}
\langle\text{SayStatement}\rangle & ::= & \texttt{SAY}\ \langle\text{MessagePattern}\rangle\ \texttt{;} \\
\langle\text{MessagePattern}\rangle & ::= & \texttt{[}\ \big(\langle\text{Attribute}\rangle\ \big(\texttt{,}\ \langle\text{Attribute}\rangle\big)^{*}\ \big)^{?}\ \texttt{]} \\
\langle\text{Attribute}\rangle & ::= & \langle\text{Identifier}\rangle\ \texttt{=}\ \langle\text{Value}\rangle
\end{array}$$

Notes:

The `SAY`-statement enforces a read-only scope.

It is a semantic error to use the same identifier in more than one attribute.

Semantics:

[8] The restriction is necessary in order to ensure that the generated code is free of unreachable statements as defined in [17] §14.20

$$\chi(\underline{\textbf{SAY}}\ [\iota_0 = \epsilon_0,\ \ldots,\ \iota_n = \epsilon_n\ \ ]\ ;\ )$$

$\equiv$

$\texttt{tempMap} = \underline{\textbf{new}}\ \texttt{HashMap}(2(n+1));$
$\texttt{tempMap.put}(\iota_0,\ \chi(\epsilon_0));$
$\texttt{...}$
$\texttt{tempMap.put}(\iota_n,\ \chi(\epsilon_n));$
$\texttt{send}(\texttt{tempMap});$

15.9 Message Rules

A *message rule* conditionally executes a block based on whether the message parameter of the procedure matches a given pattern. Message rules are not statements and hence cannot be nested; however, since the message parameter is fixed for the entire procedure, there is no reason to do so anyway.

Syntax:

⟨MessageRule⟩ ::= `WHEN` (`NOTHING` | ⟨MessageGuard⟩) ⟨Block⟩
⟨MessageGuard⟩ ::= `[` (⟨xAttribute⟩ (`,` ⟨xAttribute⟩)*)$^?$ `]`
⟨xAttribute⟩ ::= ⟨Identifier⟩ `=` (⟨Expression⟩ |
⟨Fact⟩ (`AS` ⟨Variable⟩)$^?$)

Notes:

In case an `AS`-clause is given, the variable is denoted an *alias* for the expression; an alias always occurs in a write-only scope. Otherwise, the message rule, including the block, constitute a read-write scope.

Semantics:

The simplest case is

$$\chi(\underline{\textbf{WHEN}}\ \underline{\textbf{NOTHING}}\ \beta)$$

$\equiv$

```
if (msg == null) {
    $\chi(\beta)$
}
```

In general, let $\alpha_0, \ldots, \alpha_n$ be the attributes of the guard pattern. Then,

$\chi(\underline{\textbf{WHEN}}\ [\alpha_0,\ \ldots,\ \alpha_n]\ \beta)$

$\equiv$

```
while (msg != null) {
    $\chi(\alpha_0)$
    ...
    $\chi(\alpha_n)$
    $\chi(\beta)$
    break;[9]
}
```

Consider an attribute $\iota = \epsilon$ which does not contain any variable defs. Then,

$\chi(\iota\ = \epsilon)$

$\equiv$

```
if (!msg.containsKey($\iota$)) break;
tempObj = msg.get($\iota$);
if (!isEqual(tempObj, $\chi(\epsilon)$)) break;
```

If ϵ were a variable def ξ, then

$\chi(\iota\ = \epsilon)$

$\equiv$

```
if (!msg.containsKey($\iota$)) break;
tempObj = msg.get($\iota$);
$\Delta(\xi$ , tempObj)
```

Finally, consider the case where ϵ is a fact. First, the lines

[9] The `break` will not be added if β ends with a `RETURN`-statement

```
if (!msg.containsKey($\iota$)) break;
tempObj = msg.get($\iota$);
if (!( tempObj instanceof Fact)) break;
tempFact = (Fact)tempObj;
```

will be produced. Then, in case of a ground fact, the line

```
if (!tempFact.equals($\chi(\epsilon)$) break;
```

will be added. Otherwise the fact in question will contain variable defs ξ_i for $i \in \delta_0, \ldots, \delta_n$; in this case the lines

```
tempBitSet.clear();
tempBitSet.set($\delta_0$);
...
tempBitSet.set($\delta_n$);
if (!tempFact.equals($\chi(\epsilon)$, tempBitSet) break;
$\Delta(\xi_{\delta_0}$, tempFact.getTerm($\delta_0$))
...
$\Delta(\xi_{\delta_n}$, tempFact.getTerm($\delta_n$))
```

will result. Finally, if an alias not being a wildcard variable, say ξ_α, has been specified, then the line

```
$\Delta(\xi_\alpha$ , tempFact)
```

will be produced.

Part IV

System Test

Chapter 16

System Test Strategy

Each component of the ACT has been thoroughly hand-tested during development. In order to ensure overall system correctness, however, an extensive and reproducible test setup is required. Since it is a key design objective that the ACT should serve as a tool for experiments with agent-based systems$_{2.1}$, it seems a good idea to make a combined solution to the two goals. Therefore:

- The ACT should incorporate a large-scale test bed on which the toolkit can be thoroughly tested.
- The test bed should be somewhat complex in order to discover the strengths and uncover the weaknesses of the ACT.
- The test bed application should make a relevant case for the use of agents, in order to ensure that the evaluation of the ACT is meaningful.
- The test bed should preferably allow for solutions of a complexity ranging from the very simple to the highly complex. In this way some significant benefits are gained:
    - It will be possible to provide a *demo* application based on the test bed, without making the demo unduly complex. A description of the demo can then be instructive for potential users of the ACT.
    - The test bed can be used as the basis of experiments with various more or less complex agent-based systems.

The demands on the test scenario stated above can should not be underestimated, though: with an increase in generality of the test bed, the problem of makning a relevant demo becomes harder, since more possibilities must be considered.

The choice and design of the test bed is the subject of the next chapter, while its implementation is described in chapter 18 and appendix B respectively. Chapter 19 describes the design and implementation of the demo.

16.1 Choosing a Test Bed

Diplomacy® is a board game for up to 7 players, each of which controls one of the Great Powers[1] of pre-World War I Europe; the objective of the game is to gain control of the map through diplomatic negotiations and army movements. Negotiations play a central role in the game. To quote [4],

> "Diplomacy *is a game of negotiations, alliances, promises kept, and promises broken. In order to survive, a player needs help from others. In order to win the game, a player must eventually stand alone. Knowing whom to trust, when to trust them, what to promise, and when to promise it is the heart of the game."*

The game was originally invented by Allan B. Calhamer around 1959, and is presently published by Avalon Hill Games, Inc. [4], to which the trademark is registered.

Diplomacy has been chosen as the basis of the test bed because:

- The key role of negotiation makes Diplomacy relevant in relation to the ACT, since each player is easily conceptualised as an agent.
- The game concepts are generally speaking fairly simple, although the game contains many details (the rules [4], with explanatory figures and examples, take up 24 A4 sheets). In terms of state space, however, the game is undoubtedly complex: the game contains 74 *provinces* and anywhere between 2 and 34 *units* which each occupy a different province. Two types of units exist, and each unit belongs to one out

[1] The Great Powers are: England, Germany, Russia, Turkey, Italy, France and Austria-Hungary.

of 7 players. Not every location allows every type of unit, though, and no player can own more than 18 units.

Since units move *simultaneously*, the number of possible state transitions is equally large: depending on the game situation a unit can have more than 30 possible moves, although 3–10 is more typical.

- Due to the use of negotiation and the large state space involved, the application can indeed accommodate solutions of wildly varying complexity.

In order to focus on the main issue — negotiation — the test bed is a somewhat simplified version of Diplomacy; its design is described in the next chapter.

Chapter 17

The Game of HAPLOMACY

The game of HAPLOMACY is a simplified version of Diplomacy$_{16.1}$, as the name implies[1]. The current chapter describes the basic concepts of the game, using the same terminology as used in [4].

17.1 Design Idea

HAPLOMACY mainly simplifies Diplomacy in two ways:

- HAPLOMACY uses only one kind of *unit*, where Diplomacy uses both *armies* and *fleets*. The result is a large reduction in the complexity necessary to adjudicate orders$_{B.3.1}$, but with a less significant reduction in the complexity and playability of the game.
- HAPLOMACY is an *abstraction* of Diplomacy. For example, the game board is based on a general graph data structure, rather than a map of Europe anno 1900 whose geographical characteristics make special exceptions to the movement rules. The benefit is a simpler, cleaner implementation which does not get bogged down by simulation details which are irrelevant to the application.

[1] The word 'diplomacy' derives from the Greek 'diploos', meaning *double* or *two-fold*; analogously, 'haploos' means *single* or *simple*, hence the name HAPLOMACY.

17.2 Basic Game Concepts

The game of HAPLOMACY is played by 2–7 *players*, each of which are identified by a number in the range 0–6.

Each player controls a number of *units*, which are placed in *provinces* on a *game board*, such that each province is either *vacant* or *occupied* by exactly one unit. A subset of the provinces are *support centres*, which can either be *owned* by a player or be *neutral.* The support centres which have a designated initial owner (i.e., the support centres which are not initially neutral) make up the *home country* of their designated owner.

The game is played in *turns*, identified by an integer which is initially 0. Each turn has two *seasons*, *spring* and *fall* — simply written 'S' or 'F'. Thus, the turns of a HAPLOMACY game are, in order, 0S, 0F, 1S, 1F, etc.

Each turn begins with a *negotiation phase* where each player has a chance to detect and to influence the strategy of other players, and vice-versa. There are no restrictions or requirements whatsoever on *how* this negotiation is to be performed.

Once negotiations have come to an end, each player issues *orders* to their own units, after which all orders are simultaneously revealed and their combined results adjudicated according to a set of rules. As an effect of this adjudication, it may be necessary for one or more units to *retreat*, which is handled in a separate phase. In the fall, an additional *adjustment* phase takes place; here, each player gains or loses units based on the number of support centres they own.

The winner of the game is the first player to own more than half of the support centres of the game board.

17.3 The HAPLOMACY Game Board

The HAPLOMACY game board is not a fixed design, but rather a general unweighted, undirected graph data structure, where each vertex corresponds to a province (which is either an ordinary province, or a support centre; in the latter case, the support centre may also has a designated initial owner), and each edge identifies the borderlines between provinces[2].

[2] A province is by definition not adjacent to itself.

Nevertheless, in order to make a standardised scenario for the test bed, a *default game board* has been defined. Since the default game board is treated exclusively in the remains of the document, the shorter term *game board* will be used with reference to default game board.

The default game board, which is shown in figure 17.1 below, is a 9×9 grid of provinces. Each province is identified by a single letter indicating the column (A, B, . . .) followed by a single digit indicating the row (starting from 0). Provinces are adjacent to each other both vertically, horizontally and diagonally. The game board is symmetrical, and is exclusively designed for 4 players. Each home country consists of 3 support centres, and 9 additional support centres are neutral. The grand total of support centres is thus 21, which makes 11 support centres required to win the game.

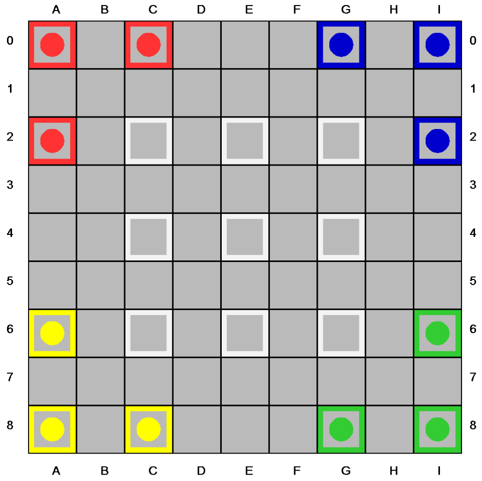

Figure 17.1: The default HAPLOMACY game board. Units are shown as circles in the colour of their respective owner; support centres are framed in the colour of their owner.

17.4 Giving Orders

The possible orders for a unit are:

- *hold*, meaning that the unit attempts to stay in the province it currently occupies.
- *move*, meaning that the unit attempts to enter a new province, which must be adjacent to the province it currently occupies. This province is said to be the *destination* of the move. If the destination is occupied, the move is said to be an *attack*, and the occupying unit is being *attacked* by the moving unit.
- *support*, meaning that the unit will holds its current position, but will combine its strength with another unit, which possibly could belong to a different player. A support is valid only if
  - The supported unit is ordered to hold and is occupying a province which is adjacent to the location of the supporting unit.

  or
  - The supported unit is ordered to move into a province which is adjacent to the location of the supporting unit[3].

In contrast to Diplomacy, there is no requirement that the order given will fail if the order does not mention the specific order of the supported unit: in HAPLOMACY, a support order is expressed simply by specifying the supported unit.

17.5 Resolving Orders

The order resolution phase is, even with the simplifications performed in HAPLOMACY, rather complex. The general principles are the following:

- Only one unit can occupy a province at a time.
- All units have equal strength.
- A unit being supported by n units has a combined strength of $n + 1$. A unit and its supports are collectively denoted an *army*. The supported unit in an army is denoted the *army leader*.
- If a unit giving support is attacked, support will be *cut*, i.e. the effect of the support will be nullified.
- If two (or more) armies of equal sizes try to occupy a province the result will be a *standoff*: all units involved will fail to move.

[3] The graph should prevent a province from being adjacent to itself to avoid a unit from supporting an attack on itself.

- If a unit is attacked by a larger army than its own, then the attacked unit will be *dislodged*: the unit fails to move, is removed from its current location and is forced to *retreat*.

Due to the large amount of material involved, the full adjudication procedure is described in appendix B.

17.6 Retreats

The units which were dislodged as the result of the movement phase are forced to *retreat*, that is, to move to a province which fulfils the following criteria:

- The province must be adjacent to the province from which the unit was dislodged.
- The province must be vacant.
- The province cannot have been left vacant as the result of a standoff.

If no such province exists the dislodged unit will be *disbanded*, i.e., it will be removed from the game.

The units which are not disbanded are given a movement order to one of the eligible provinces. Contrary to the case in Diplomacy, the choice of province is performed automatically following a fixed procedure: the province which minimises the sum of distances to the provinces of the unit's home country (using half the sum of distances in case the province is part of the home country), making an arbitrary choice in case of a tie.

After all dislodged units have been given a movement order, the moves are resolved as usual$_{17.5}$. In case two or more retreating units are standing each other off, they will all be disbanded.

17.7 Adjustments

After a fall turn has been executed, ownership of the support centres of the board is established, using the following rules:

- A province which is vacant at the end of a fall turn retains its current owner, regardless of events in the spring turn.

- Ownership of a province which is occupied by a unit at the end of a fall term is transferred to the owner of the occupying unit.

Once ownership of the support centres has been established, the number of units owned by each player will be adjusted to match the number of owned support centres. Adjustments are carried out as follows:

- A player who has lost control of all support centres is eliminated from the game.
- A player who has control of more than half of the support centres will be declared the winner of the game.
- A player who owns more units than support centres will be forced to disband the excess units. The units are disbanded by repeatedly selecting the unit the farthest from its home country, that is, the unit whose location maximises the sum of distances to the provinces of the unit's home country, making an arbitrary choice in case of a tie.
- A player who owns more support centres than units may place extra units in the unoccupied support centres of its home country, but nowhere else; it may therefore be the case that a player begins a spring turn with more support centres than units, but not vice-versa. Build locations are selected automatically, using an arbitrary choice between the eligible support centres.

Chapter 18

The Haplomacy Test Bed

A simulator for the Haplomacy game which has been introduced in the preceding and whose full set of rules are specified in appendix B, has been implemented as part of the `demo` package in the ACT. The implementation, which is denoted the *test bed*, makes a standardised test scenario which fulfils the requirements stated in chapter 16.

The UML class diagram in figure 18.1 on the next page shows the overall structure of the Haplomacy test bed implementation. The various classes are briefly described in the following.

18.1 The `HaplomacyBoard` Class

The class `demo.HaplomacyBoard` represents the game board itself and its *state*, that is, the location of units and their orders; this information is contained in the `NeighbourGraph` instance. The class can be instantiated with a custom-made game board if desired, but it defaults to using the default game board shown in figure 17.1 on page 133.

In addition to the above, the `HaplomacyBoard` contains a pre-calculated distance matrix for the graph (which does not change during the course of the game) in order to provide this information (which is used repeatedly during the retreat and build phases) as quickly as possible. Apart from the cached distance matrix, the `HaplomacyBoard` also provides functionality

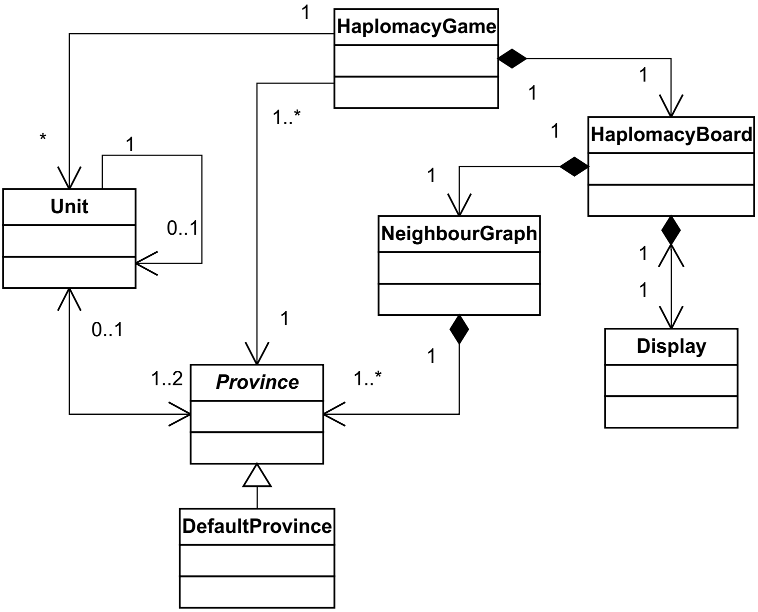

Figure 18.1: Structure of the test bed implementation.

to visualise (and to customise the visualisation of) the game board itself, using the `Display` class described below.

18.1.1 The `Display` Class

The class `Display` is an inner class in `HaplomacyBoard`. The class is itself a specialization of `javax.swing.JComponent`, and it provides a visualisation of the entire game board. Figure 17.1 on page 133 as well as the various figures in appendix B have all been created from the output of the `Display`.

18.2 The `NeighbourGraph` Class

The class `util.NeighbourGraph` is a generic data structure representing an arbitrary unweighted graph, using a standard neighbour-list representation internally. In addition to the representation-related methods the class provides functionality to calculated shortest-path distances.

The `NeighbourGraph` allows any `Object` instance to represent a vertex; when used in the test bed, however, the vertices are assumed to be `Province` instances.

18.3 The `Province` Class

The abstract class `demo.Province` represents a single province in the game board, and specifies the methods necessary to visually render the province, and implements a set of methods to maintain its internal state — the unit which optionally occupies it.

18.3.1 The `DefaultProvince` Class

The `DefaultProvince` is used in the default game board. It specialises the `Province` class by providing the code necessary to visually render the province as a single square (and optionally a row/column label) at a location specified to its constructor.

18.4 The `Unit` Class

The `Unit` represents a single unit. Its internal state consists of its *location* (a `Province`), its *destination* (the `Province` to which the unit optionally has been ordered to move) and the `Unit` which it optionally has been ordered to support.

In addition to the above, the `Unit` contains code to render itself and to show its current orders, c.f. the diagrams in appendix B.1.

18.5 The `HaplomacyGame` Class

The class `demo.HaplomacyGame` represents an entire game of HAPLOMACY. It internally uses a `HaplomacyBoard` to represent and visualise the game board, but also maintains separate lists of the units and support centres.

The main idea of separating the `HaplomacyGame` from the `HaplomacyBoard` is to separate the static and the dynamic behaviour of the game: `HaplomacyBoard` is a data structure to represent the game state, while `HaplomacyGame` is responsible for the dynamic behaviour of the game, i.e. for executing a single turn of the game[1].

The method `update` in `HaplomacyGame` transforms the state of the game into a new state corresponding to the advancement of a single turn in the game. This transformation is carried out according to the general principles in sections 17.2 and 17.5, and the specific rules in appendix B; hence the `HaplomacyGame` implements an *adjudicator* which uses the algorithm specified in appendix B.3.1. The computational complexity of that algorithm, as implemented in the `HaplomacyGame`, is briefly described in section 18.5.3.

In addition to adjudication of orders, the `HaplomacyGame` provides two different simple strategies for giving orders; these are described in sections 18.5.1 and 18.5.2.

18.5.1 A Defensive Strategy

The method `giveDefensiveOrders` in `HaplomacyGame` gives orders to the units of a player following the simple strategy described in the following.

In addition to the player ID the method takes two parameters: a set of *friendly players* and a set of *neutral provinces*, which influence the orders in certain situations. For each unit owned by the player its orders are given according to the following rules:

- If the unit is occupying a support centre owned by the player, then order that unit to hold it.
- Otherwise, if the unit is adjacent to a province which is not neutral and which is part of the player's home country then
    - If the province is occupied by one of the player's own units, then support that unit.
    - Otherwise, if the province is not occupied by a friendly player, then order the unit to attack the province
- Otherwise, locate the support centre nearest to the unit under the following restrictions:

[1] It is assumed that orders — which are represented in the `Unit` instances of the game — have been specified at an earlier time.

- The support centre cannot be neutral.
- The support centre cannot be occupied by a friendly player.
- If the support centre is owned by the player it must be empty.
- If the support centre is not owned by the player it must be part of the player's home country

Then order the unit to the chosen support centre, if it is adjacent to the unit, and otherwise to an adjacent province en route to that support centre, such that the following criteria are fulfilled:

- The destination province cannot be neutral[2].
- The destination province cannot be occupied by a friendly player.
- The destination province should preferably be empty. A province which is occupied (either by the player itself or by a player which is not considered friendly) should be selected only if no other possibility exists[3].
- In case no province fulfils the criteria the unit is ordered to hold.

18.5.2 An Offensive Strategy

The offensive strategy — which is executed by the `giveOffensiveOrders` method shares many aspects of the defensive strategy$_{18.5.1}$; the strategy also uses a set of friendly players and neutral provinces, in addition to a parameter defining the *strength* of the attack: if the player controls n units, then an attack of strength σ will encompass $\lfloor n \cdot \sigma \rfloor$ units, while the remaining units will be ordered to hold.

The strategy is executed by first finding a *target* support centre, which the attacking units will try to occupy. The target is subject to the following restrictions:

1. The target support centre cannot be neutral.
2. The target support centre cannot be occupied or owned by a friendly player.
3. The target support centre cannot be owned by the player itself.

Among the eligible support centres (if any) the target is found in the following way:

[2] Once a unit has been ordered to a certain province, that province is considered to be neutral — otherwise the player's own units would cause a standoff

[3] It is generally not sufficient to attack an occupied province without support — the order will not have any effect unless the occupant moves by itself.

- If an eligible support centre is occupied by the player, then order the occupying unit to hold and select that support centre as the target.
- Otherwise, choose the eligible support centre which minimises the sum of distances to the player's units. If at least one of the support centres of the player's home country was eligible, then the target is selected only among these support centres[4].

In case no target could be found, then the strategy *fails*, i.e., no orders will be given; the return value of the `giveOffensiveOrders` method will indicate whether this is the case.

If a target was found, then the specified number of units will be given orders in the following way:

1. Sort the units by their distance to the target.
2. Select the units to participate from the sorted list of units, ignoring the units for which it is impossible to participate due to the restrictions (friendly units and neutral provinces). Each participating unit is given a movement order as described at the end of section 18.5.1; if more than one unit were adjacent to the target, then only one will attack the target while the others will support it.

18.5.3 Complexity of the Adjudication Algorithm

The adjudication algorithm implementation relies on fast access to certain data[5], which are maintained separately by the `HaplomacyGame`. Additionally, the algorithm builds extra data structures[6] during the validation of orders. For a game containing n units, this initial phase can be executed in $O(n)$ time, with an additional memory consumption of $O(n)$.

For each step in the algorithm, every unit not ordered to hold will have to be examined at least once (but not more than a few times in total). Due to the data structures which are maintained during the algorithm (of which only operations of (amortised) constant time are used), the asymptotical computational complexity is bounded by the number of times the algorithm

[4] It is generally better to try to regain control of the home country than to occupy a new support centre, since that leaves more possible locations for new units to be built.

[5] The relevant data include various subsets of the units and provinces of the `HaplomacyBoard`.

[6] The additional data structures — which are updated during each step of the algorithm — map units to their supports, attackers etc.

is run, multiplied by the number of units examined in each run. Since every step of the algorithm is guaranteed to change the orders of at least one unit to a hold order, the worst case running time of the entire algorithm is therefore

$$O(n) + \sum_{i=n}^{1} O(i)$$

which amounts to

$$O(n^2)$$

Since the number of units is bounded by the number of support centres in the game (which is 21 in the default game board), a computational complexity of $O(n^2)$ should not present a problem in practice. Furthermore, the algorithm generally provides a better performance than $O(n)$ — it is actually $\Omega(n)$.[7]

[7] The algorithm generally removes the largest possible number of units from consideration at each step. Moreover, at the first step in the actual implementation, *every* eligible unit (a unit ordered to move into an empty province which is not targeted by any other unit) is found and moved in one step, requiring only $O(n)$ total. In the best case this bounds the running time of the entire algorithm.

Chapter 19

Playing HAPLOMACY With AGENTC

With the test bed$_{18}$ being established, the subject of the *demo* is fairly obvious: the demo should be simulated HAPLOMACY game played by agents constructed by means of the ACT.

What is missing in order to reach the goal will then be the following:

1. A communicative constituent and a reactive constituent for the player agents$_{??}$.
2. A suitable set of extension modules which allow the agents to interact with the game simulation$_{??}$.
3. A component which controls the game simulation$_{19.3}$.
4. Definition of the attitudes and facts to be used in the player programs$_{19.4}$.
5. A protocol for agent communication$_{19.5}$.
6. ACMEs which can control the agents, i.e., the players$_{19.6}$

All classes relating to the demo have been collected in the `demo.package` of the ACT.

19.1 The `DemoAgent`

The `DemoAgent` provides a simple agent implementation. The class extends `util.MessageController`$_{8.1.4}$, and it distributes messages to the appropriate methods of the `DemoAcme`$_{19.2}$, based on special control messages from the game simulation$_{19.3}$. The `DemoAgent` therefore serves as a combined communicative and reactive constituent.

19.2 The `DemoAcme`

The `DemoAcme` is the implementation basis for the ACMEs which represent the four players. The class overrides some of the methods of its superclass `afc.Acme`$_{5.1}$ such that its actions can be logged; this is useful when analysing the behaviour of the ACMEs. Equally important, though, is that the `DemoAcme` specifies a set of procedures$_{13.4}$ whose signatures are inherited$_{7.1.4}$ by the AgentC code.

The inherited code is described in sections 19.2.1 and 19.5. The `DemoAcme` relies on the extension modules which are described in sections 8.1.3, 19.2.2 and 19.2.3. Appendix C lists the AgentC source code for the set of ACMEs used in the system test; these all implicitly inherit from the `DemoAcme`, which must be specified to the ACC as their superclass.

19.2.1 The `DemoAcme` Signature

By specifying `DemoAcme` as the superclass of the output class, the ACC will implicitly$_{7.1.4}$ produce the AgentC code

```
DEFS {
    $RED = 0;
    $BLUE = 1;
    $GREEN = 2;
    $YELLOW = 3;
}

PROCEDURE init();
PROCEDURE negotiate();
PROCEDURE giveOrders();
PROCEDURE updateStatus();
```

in addition to the code inherited from the `DemoProtocol` interface$_{19.5}$ implemented by the `DemoAcme`.

The four symbol definitions above correspond to the identity codes used by the four different ACMEs, and to their player numbers in the game.

The initialisation procedure `init` is called once by the `DemoAgent`$_{19.1}$ constructor, allowing the ACME to perform special initialisation code.

The procedure `negotiate` is the main procedure of the ACME; it is called once for each message received during the simulation's negotiation phase$_{19.3}$, and is also called with a certain timeout if no messages are received. The AgentC code will therefore typically contain series of message rules matching a distinctive message pattern, which respond appropriately to incoming message based on the ACMEs beliefs about the player who sent the message. Additionally, the procedure will most probably contain a single

```
WHEN NOTHING {
    ...
}
```

message rule where the ACME determines which new messages to be sent, by investigating the knowledge base and using the queries provided by the `DemoInvestigator`$_{19.2.2}$. If this rule is the only part of the procedure which sends new messages, the timeout period used by the `DemoAcme` effectively limits how many new negotiations the ACME will start, and hence also limits the processing time necessary to simulate the negotiation phase.

The procedure `giveOrders` is called once by the `DemoAcme` after negotiations have ended. The procedure conveys the ACMEs intentions to the simulation by issuing one or more of the actions defined in the `DemoActuator`$_{19.2.3}$.

The procedure `updateStatus` is called by the `DemoAcme` once for every message sent by the game simulation after orders have been adjudicated, and an extra, final time with `NOTHING` as parameter. This allows the ACME to adjust its beliefs about the other players by judging their actions, and to pre-calculate its goals for the next negotiation phase.

19.2.2 The `DemoInvestigator`

The `DemoInvestigator` provides a series of queries which can be used to make simple player strategies. The queries provided are:

`add(x, y)`
Calculates the sum of the two numeric or string operands.

`sub(x, y)`
Calculates the difference between the two numeric.

`mul(x, y)`
Calculates the product of the numeric operands.

`div(x, y)`
Calculates the quotient of the numeric operands.

`random(x)`
Yields a random number in the range $[0, x)$.

`random()`
Yields a random number in the range $[0, 1)$.

`strongestPlayer()`
Determines the number of the player currently having the most units.

`strongestOpponent()`
Determines the number of the player currently having the most units, without considering the player controlled by ACME itself.

`weakestPlayer()`
Determines the number of the player currently having the least units.

`weakestOpponent()`
Determines the number of the player currently having the least units, without considering the player controlled by ACME itself.

`strengthOf(x)`
Determines the strength of the specified player, relative to the other players.

19.2.3 The `DemoActuator`

The `DemoActuator` is used by the ACMEs to place orders for their respective units in the game simulation. The `DemoActuator` defines three actions:

`print` Prints the concatenation of the parameters, followed by a line-break character, to `System.out`. In contrast to the other actions provided, the `print` action behaves specially in the sense that no fixed number of parameters is required — *any* number of parameters is accepted.

`registerFriend(x)`
Registers that the player `x` is considered to be friendly.

`unregisterFriend(x)`
Removes the registration of player `x` being friendly.

`resetFriends()`
Resets the internal list of registered friendly players.

`defend()`
Gives the player's units a set of defensive orders$_{18.5.1}$; the previously registered friendly players influence how these orders are given.

`attack(x)`
Gives the player's units a set of offensive orders$_{18.5.2}$, using `x` percent of the player's units in the attack. The previously registered friendly players influence how the orders are given. In case no suitable support centre could be chosen for the attack the action will fail, i.e., no orders will be given, and the action statement will, when evaluated as a condition, not be fulfilled.

19.3 The `HaplomacyDemo`

The `HaplomacyDemo` class implements a Haplomacy game simulation, using the default game board; the only arguments to its constructor are the four `DemoAcme` instances which should be used to control the four players — the necessary `PostOffice`, `DemoAgents`, their extension modules and the `HaplomacyGame` instance will then be created automatically.

The main purpose of the `HaplomacyGame` is to provide a centralised coordination of the agents, such that the negotiation and order writing phases are clearly separated; in addition, the `HaplomacyGame` sends status messages to the player agents after orders have been adjudicated.

The simulation has 6 internal states called *phases*; the simulation progresses by switching from one phase to the next in a cyclic fashion. Three of the phases are *active* in the sense that a number of agents are potentially executing in parallel, while the remaining phases are *passive*, meaning that none of the agents are executing. The simulation alternates between a passive and an active phase; the purpose of the passive phases (which may seem superfluous at a first glance) is to provide breakpoints where the internal state of the simulation is stable, such that it can be examined. The phases used by the simulation are the following:

1. Beginning of turn (passive)
2. Negotiation (active) — all agents execute in parallel[1].
3. End of negotiation (passive)
4. Order writing (active) — each agent executes in parallel[2].
5. End of order writing (passive)
6. Adjudication (active) — the `HaplomacyDemo` adjudicates the given orders and sends status messages to the player agents[3].

The `HaplomacyDemo` class both provides a method to advance the simulation phase one step at a time, and a method to play one or more entire game turns at once. It even defines a method to play one or more entire game simulations; this method is the basis of the system test described in chapter 20

19.4 Using the Knowledge Base

For the simple protocol$_{19.5}$ and player strategies$_{18.5.2}$ the attitudes specified by

```
ATTITUDES {
    #DID = 0;
    #BELIEVE = 1;
    #B = 1;
    #INTEND = 2;
    #I = 2;
}
```

are more than sufficient. Only two specific types of facts have been given a fixed meaning common to all player ACMEs, as defined below; however, some of the player ACMEs may also use the attitudes in their internal implementation.

[1] The `DemoAgents` continuously call the respective `negotiate` procedures for each incoming messages, or after a given timeout

[2] The `DemoAgents` call their respective ACMEs' `giveOrders` procedures once.

[3] Each `DemoAgent` invokes their respective ACMEs' `updateStatus` procedures once for each status message sent, and a final time with `NOTHING` as parameter.

`#B relation(?X, ?Y, ?r)` represents the agent's belief that the player relation$_{19.4.1}$ between `?X` and `?Y` has value `?r`.
`#I relation(?X, ?l, ?u)` is used in communication to represent the sender's intention that the receiver's player relation towards `?X` should be in the range [`?l`, `u`].

19.4.1 The Player Relation

The *player relation* `#B relation(?X, ?Y, ?r)` determines how player `?X` treats player `?Y`. The value used in the player relation is a floating-point value in the range $[-1.0, 1.0]$; 0.0 represents a neutral relation, a value larger than 0.0 denotes a friendly relation, while a value less than 0.0 indicates a hostile relation.

When giving orders the ACMEs use their beliefs about the player relation to determine which players are eligible for an attack, and which should be left alone.

The initial value of the player relation is determined by the specific ACME, which also determines how negotiations and actions should influence the value.

Although the form of the player relation generally allows beliefs about other player's beliefs to be represented, the demo ACMEs solely store beliefs about their own relations, meaning that `SELF` will be stored in the first term.

19.5 Communication Protocol

The choice of communication protocol in the demo defines how and about what the agents can communicate. In order to keep the implementation as simple as possible the communication protocol is itself rather rudimentary.

19.5.1 Message Structure

The demo uses a GMI message structure. The following fixed attribute names are used[4]:

[4] The strings are defined by the `DefaultMessenger`$_{8.1.3}$ class.

`to` Holds the ID of the ACME to which the message is addressed.

`from` Holds the ID of ACME which sent the message, or the special constant `simulation`, if the message was sent from the game simulation$_{19.3}$.

`type` Holds a string constant which identifies the type of message[5]; these constants are the sole contents of the `DemoProtocol` interface, from which the `DemoAcme` inherits; hence the constants are directly usable in the AgentC code$_{7.1.4}$.

`contents` Holds an AgentC value whose interpretation depends on the `type` of the message.

19.5.2 Message Contents

The message content language is the language of AgentC values$_{14}$.

19.5.3 Message Semantics

The `DemoProtocol` interface defines a set of `String` constants which can used as `type` specifications. The constants themselves, and their prescribed usage which is defined in table 19.1 on the facing page, constitute the message semantics.

19.5.4 Message Protocol

The message protocol defines when and how a message can legally be sent.

Simple as it is, the communication protocol allows only two kinds of message exchanges:

1. A *notification*, meaning that a single message which cannot be answered is sent.
2. A *conversation*, meaning that some message is sent, to which the recipient is obliged to answer in a given way. The message initiating the conversation is known as a *request*, while the message which ends the conversation is known as a *reply*.

[5] The type is the equivalent of a performative in KQML. The word *type* is preferred in the demo because it is easier to type in the AgentC program.

`type`	`contents`	**Description**
`UNIT_ATTACKED`	An agent ID	Sent by the simulation to indicate that the receiver was attacked by the given player.
`SUPPORT_CENTRE_ATTACKED`	An agent ID	Sent by the simulation to indicate that a support centre owned by the receiver was attacked by the given player.
`SUPPORT_CENTRE_CONQUERED`	An agent ID	Sent by the simulation to indicate that a support centre originally owned by the receiver overtaken by the given player.
`PLAYER_ELIMINATED`	An agent ID	Sent by the simulation to indicate that the given player was eliminated from the game.
`REQUEST`	`#I relation(?X, ?l, ?u)`	The sender requests that the relation of the recipient towards player `?X` should be in the range $[?l, ?u]$.
`ACCEPT`	...	The sender accepts a previous request, retransmitting the contents.
`REJECT`	...	The sender rejects a previous request, retransmitting the contents.

Table 19.1: Semantics of the message types defined in the `DemoProtocol` interface

The overall rules set by the message protocol are the following:

1. A request must be answered in the way prescribed in table 19.2 under here
2. An agent should reply to a received request as soon as it is able to do so.

The possible notifications are the following:

1. `UNIT_ATTACKED`
2. `SUPPORT_CENTRE_ATTACKED`
3. `SUPPORT_CENTRE_CONQUERED`
4. `PLAYER_ELIMINATED`

None of these may be sent by a player agent — the notifications are used exclusively by the simulation to notify agents of results.

Table 19.2 lists the two possible conversations. No other conversations are legal.

Request	`[from=?S,to=?R,type=REQUEST,contents=#I relation(?X,?l,?u)]`
Reply	`[from=?R,to=?S,type=ACCEPT,contents=#I relation(?X,?l,?u)]`
Request	`[from=?S,to=?R,type=REQUEST,contents=#I relation(?X,?l,?u)]`
Reply	`[from=?R,to=?S,type=REJECT,contents=#I relation(?X,?l,?u)]`

Table 19.2: Conversation protocol

19.6 Haplomacy Player ACMEs

Four different ACMEs have been defined; their full AgentC source code, with rich comments, can be found in appendix C. This section provides only a brief description of the differences in the way these players negotiate. Common to all players is that they specify a different initial player relation, that they reduce player relations towards an attacking player, and that they give offensive orders$_{18.5.2}$ with a fixed strength, subject to a minor random modification.

19.6.1 Characteristics of the Ruthless Player

The ruthless player does not initiate new conversations. The only activity which the ruthless player carries out on its own initiative during negotiations is to make a slight random adjustment to its player relations.

The ruthless player is accepts requests from the players which are stronger than the ruthless player itself.

When giving orders the ruthless player attacks any player towards whom the player relation is below 0.0. In case a no such player exists, the player with whom the ruthless player has the weakest player relation will be attacked, regardless of how friendly the actual player relation is.

19.6.2 Characteristics of the Vindictive Player

The vindictive player never initiates an attack by itself. Once a player has attacked a support centre owned by the vindictive player, that player will risk an attack by the vindictive player 50% of the time if the player relation is friendly, and every time otherwise.

During negotiations the vindictive player requests from every friendly player to lower their player relation towards the enemies of the vindictive player.

The vindictive player accepts requests from other players, but cannot be persuaded to raise the player relation above 0.0 for an enemy.

19.6.3 Characteristics of the Cautious Player

When attacked by a player the cautious player subsequently negotiates by sending the attackers a requests to have friendly player relations towards the friendly player. If the request is rejected the cautious player will reduce the player relation towards that player.

The cautious player readily updates its player relation when requested, although the player relation is not set lower than 0.0 as the result of a request.

For each of the players which did not attack the cautious player, the player relation will be increased slightly every turn.

19.6.4 Characteristics of the Cowardly Player

The cowardly player negotiates by requesting all players stronger than itself to have friendly player relations towards it.

The cowardly player accepts requests from players stronger than itself, unless the request is to reduce the player relation towards an even stronger player.

If the cowardly player only has neutral or friendly relations with the other players it will attack the weakest player.

For each turn the cowardly player slightly increases its relations towards players stronger than itself.

Chapter 20

System Test Results

Apart from a thorough hand-inspection to verify that the `HaplomacyDemo` and the player ACMEs (and hence the ACT) perform as expected, an extensive, automated system test has been carried out; its results are presented here.

The extensive system test not only ensures that the implementation is free of serious errors, but it also shows some interesting aspects of the Haplomacy game and last but not least about the ACME players.

20.1 Test Scenario

The default game board was used, with the fixed player allocation defined below:

Red	**(0)**	The ruthless player.
Blue	**(1)**	The vindictive player.
Green	**(2)**	The cautious player.
Yellow	**(3)**	The cowardly player.

One thousand (1000) Haplomacy games were simulated in this setup, using the `playGames` method in `HaplomacyDemo`, with the following parameters:

`maxTurns` The maximal number of turns simulated in each game was 250.

`negotiateDelay` 30 milliseconds were used for the negotiation phase.

`negotiateRate` Negotiations were performed with an interval of 5 milliseconds.

20.2 Test Results

First of all, the test shows that the ACT in general and AgentC in particular can be used to develop an agent-based solution to an application of small-to-medium size.

Second, the test shows that the ACT implementation was capable of successfully simulating 1000 different Haplomacy games without serious errors (exceptions); the total number of game turns (and hence negotiation phases) were 141665.

Third, the test shows that the ACT implementation is reasonably efficient: the total computer time used to simulate the 141665 game turns was roughly 26 seconds[1], or about 184 microseconds per simulated turn on average[2].

Fourth, the test yielded the results shown in table 20.1 on the facing page. A discussion of the data is given in chapter 21.

[1] The timings were obtained on Java `1.4.0-b92` run on an AMD Athlon 1GHz CPU

[2] Due to the simplicity of the demo ACMEs, at most four facts which are similar to each other will be present in the knowledge base at any time; in a full-scale application, efficiency would diminish accordingly.

	Won[a]	**Lost**[b]	**Undecided**[c]	**Eliminated**[d]
Red	142	330	171	357
Blue	273	351	358	18
Green	89	465	352	94
Yellow	135	413	353	99

[a] Number of games the player won

[b] Number of games the player played to the end without winning

[c] Number of games where the player participated for 250 turns without a winner being found. In total, 361 games were undecided.

[d] Number of games where the player was eliminated before the game ended

Table 20.1: System Test Results

Part V

Conclusion

Chapter 21

Discussion

This chapter provides a summary of the contents of this thesis, gives a discussion of some key issues (including the fulfilment of system requirements$_2$), and gives a brief account of future work.

The reader should be advised that in this chapter the special references to system requirements (e.g. a reference shown as$^{\#1}$) actually indicate that the author claims that the system requirement in question bas been fulfilled (or very nearly fulfilled). A reference to a system requirement without the implicit claim of its fulfilment would be in the style of a normal textual reference; in the above case the reference would be e.g. ‘See requirement 1’.

21.1 Design Objective

This thesis does indeed describe a toolkit (the ACT), and also provides a Java$^{\#3}$ implementation of the toolkit, which can be found in both binary and textual form in appendix D. The question of whether or not requirement 1 has been fulfilled hence reduces the question of whether or not the ACT fulfils the design objective as stated in section 2.1. A more detailed analysis is required to give an answer; however, it should be apparent from the descriptions found in part II, IV and appendix D that the ACT indeed provides many generic components which readily lend themselves to further specialisations$^{\#2}$.

Does the ACT facilitate implementation of agents?

Assuming that the HAPLOMACY players described in chapter 19 indeed are agents, then the answer is definitely affirmative — the ACT does actually facilitate the construction of such agents to a very high degree: In order to provide a new HAPLOMACY player agent all that is necessary is to provide the relevant ACME, which can be constructed directly from a high-level AGENTC specification by means of the ACC.

Are the HAPLOMACY players [intelligent] agents?

The definition of an [intelligent] agent is given in chapter 1; with a slight paraphrase of this definition the key question now is the following:

Q Are the players "*computer systems that are capable of [flexible] autonomous action in some environment in order to meet their design objectives*"?

A With omission of the word 'flexible' an affirmative answer is undeniable.

Q Are the players conceptualised and/or implemented in terms of mentalistic notions such as beliefs, capabilities etc.?

A The players can to some degree be said to be conceptualised and implemented in terms of mentalistic notions, since the contents of their knowledge base which is the basis of their operation is specified in such terms. However, it is questionable how deep the relationship is, and what is gained by using the mentalistic notions in the given case: it is quite obvious that the same behaviour could have been obtained without the use of mentalistic terms, since their only effect is to distinguish between different kinds of data.

The word 'flexible' induces the following questions:

Q Do the players honour the requirement that "*intelligent agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives*"?

A Yes. When the player is attacked by another player the game simulation produces a notification, to which the player will react at the first given opportunity. The same applies to requests sent from another player.

Q Do the players honour the requirement that "*intelligent agents are able to exhibit goal-directed behaviour by taking the initiative in order to satisfy their design objectives*"?

A Some of the players do indeed take initiative by engaging in new conversations during negotiations (the ruthless player does not, though). But goal-directed? It is a ultimately matter of definition, although it is unquestionable that the players are quite limited with regards to their planning capabilities (please refer to the end of section 21.2 for an explanation).

Q Do the players honour the requirement that "*intelligent agents are capable of interacting with other agents (and possibly humans) in order to satisfy their design objectives*"?

A The players certainly do interact with each other (and with the proper interface they would be able to interact with humans, too).

Does the ACT aid the design of agent-based systems?

Hopefully so. The test bed$_{18}$ is indeed a very general scenario which is well suited for such experiments. The importance of providing such a scenario can hardly be overestimated, because it frees the user from having to deal with the details of building system infrastructure, allowing focus to be placed on the agents themselves. In the words of [38]:

> *"One of the greatest obstacles in the way of the wider use of agent technology is that there are no widely-used software platforms for developing multi-agent systems. Such platforms would provide all the basic infrastructure (for message handling, tracing and monitoring, run-time management, and so on) required to create a multi-agent system. ... By the time these libraries and tools have been implemented, there is frequently little time, energy, or enthusiasm left to work either on the agents themselves or on the cooperative/social aspects of the system."*

The main motivation behind producing the ACT actually is to provide a tool which in simpler cases helps to solve the problem mentioned above — a problem which the author experienced all too well during his midterm project. Ironically — but not surprisingly — the ACT has fallen prey to the very problem it attempts to solve: The demo is not by any standards particularly advanced. But implementation of the platform (in casu the test bed and to a lesser extent the ACC and AFC) has required all available time, despite the fact that a great deal of details have been removed from the implementation at any given opportunity.

The question of whether the ACT fulfils system requirement 1 cannot be easily answered, though, since the answer of whether the ACT is *useful* or not is not easily answered due to fact that only very experiments were carried out; no serious shortcomings were uncovered, but the system test is indecisive on this point. All things considered, though, system requirement 1 is at least nearly fulfilled.$^{(\#1)}$

21.2 System Requirements

A few system requirements have already been discussed in the following. All but one of the remaining requirements are easily validated:

- The ACT provides a set of Agent Foundation Classes (the AFC) which allow virtually any kind of intra-agent architecture$^{\#4}$.
- The reactive constituent has maximal freedom to determine how it will handle events and whether to modify or inspect the deliberative constituent$^{\#5}$.
- The ACT allows an ACME to be built directly$^{\#6}$.
- The ACT defines a Generic Message Interface which can be used to transfer arbitrary messages$^{\#7}$. Additionally, the ACT provides the components necessary to build a communicative constituent based on GMI messages directly$^{\#7}$.
- The ACT includes AgentC, an APL which can express the conduct of an ACME$^{\#9}$
- The AgentC APL defined in this document allows the behaviour of ACMEs to be specified, and for input programs to be directly compiled into an ACME having the specified behaviour$^{\#11}$.
- The ACC which is part of the ACT can be used to directly compile AgentC code into equivalent Java source code$^{\#12}$.

The only system requirement which has no been mentioned yet is system requirement 10. While the requirement that AgentC is to be based on some ideas of Agent-0 and should use a simple logical language certainly are fulfilled, one problematic area remains: "AgentC *should incorporate* *as wide a range of the features of* PLACA *[32], [33] (most notably planning capabilities) as possible.*" Surprisingly, it has turned out that it is not possible to use the features of PLACA because its definition [32] *is not available* (c.f. the footnote to the entry of [32] in the list of references for an explanation). Hence system requirement 10 is actually fulfilled$^{\#10}$, although not in the way it originally was intended.

21.3 Interpretation of System Test Results

Some insteresting statistics have been gathered during the system test described in chapter 20; these findings are summarised in table 20.1 on page 159.

The results show certain subtle points about the HAPLOMACY game and test bed, and about the player ACMEs. The points are not obvious from the data, but the statistics do however fit nicely with the general trends which were observed during a closer inspection of a series of simulated games.

21.3.1 Undecided Games

As many as 36.1% of the simulated games did not have a winner after 250 turns. It is quite possible that some of these would have declared a winner if they were allowed to run a bit longer, but it is not likely that a large fraction of the prematurely terminated games would do so. The reason is that *stalemate* situations occur, i.e. situations where no progress is made because the players, due to their simple order giving strategies$_{18.5.2}$[1] and limited negotiation skills, fail to advance their positions due to standoffs. It has been seen multiple times, however, that the game does not come to a standstill, but that the game state will cycle with a period of 2–4.

21.3.2 Winning Strategies

What the statistics do not show is the *strength* (percentage of the units) with which each player attacks. With the simple order giving scheme

[1] Actually the order giving routines use explicitly randomises the data from which an arbitrary choice is made, in order to minimise the possibility of a stalemate situation, there are times when the strategy will not have more than one possible outcome

used$_{18.5.2}$, it generally pays to use a strong attack, since that will maximise the probability of gaining a new unit, for two reasons:

- The chance of succeeding in an attack is better when more units participate in the attack. This increases chances that a new unit can be gained, which can be used in a new attack, etc.
- The chance of gaining a new unit is higher if the support centres of the player's home country are left vacant — which is more likely if more of the player's units participate in the attack. It can be fatal, however, to lose one of these support centre, since this will severely reduce chances of gaining new units.

The attack strengths used by the four players are slightly randomised in order to result in less predictable behaviour. Their individual values were hand-tuned to provide a reasonable equal distribution of winners, among the four players[2], while still being consistent with the 'personality' of the player. The possible ranges of values are defined below (the values are drawn from a uniform distribution in the given range):

Red	$[0.50, 0.70)$
Blue	$[0.60, 0.75)$
Green	$[0.40, 0.60)$
Yellow	$[0.50, 0.65)$

Part of the explanation why the vindictive player (Blue) wins consistently more games than the rest of the players is therefore that the player has the highest average attack strength.

21.3.3 Analysis of the Ruthless Player

The ruthless player is eliminated far more often than any other player. This is not surprising, since the player readily makes new enemies. In the cases where the enemies are friendly towards each other the ruthless player will quickly be eliminated. However, if the enemies of the ruthless player are at war with each other, the ruthless player may actually benefit from the situation because it will have the shortest distance to its next target.

[2] The statistics show that the distribution is somewhat skewed, but this was not realised until after the system test was analysed.

21.3.4 Analysis of the Vindictive Player

The vindictive player clearly wins the most games, and is very seldomly eliminated. Three factors count to the advantage of the vindictive player:

1. The player does not make enemies with anyone unless it has been attacked. This reduces the number of enemies of the player, which makes it all the more effective when striking back.
2. The player asks its friends to attack its enemies. While the ruthless player ignores the request and the cowardly players only accepts it when the vindictive player is the strongest, the cautious player will accept it if relations are friendly; and they usually are, because the cautious player is — well, *cautious*.
3. The customary friendly relations between the vindictive and cautions players counts to the advantage of the vindictive player if the ruthless player is eliminated early in the game — the vindictive player will then quite likely be on a crusade against the yellow player, eventually eliminating it because the cautious player does not attack any player unprovoked, and because the vindictive player attacks with more strength.

21.3.5 Analysis of the Cautious Player

The cautious player has the sad record of winning the fewest games; it is simply too nice — it readily improves relations if an enemy requests it — which the cowardly player does all the time. But the low attack strength also plays a role here. The cautious player does not make new enemies on its own, which may explain why it manages to avoid elimination in many cases.

21.3.6 Analysis of the Cowardly Player

The cowardly player obtains fairly average results. The reasons are — as already mentioned, that it can exploit the gullible cautious player. And the cowardly player happily attacks the weakest player, which can pay off in certain situations.

21.4 Future Work

Since the the ACT adheres to the *‘keep it simple, stupid¡* principle, an abundancy of future work can readily be pointed out. It is worth noting, however, that the demo$_{16}$ application — which is applied to a quite complex problem — did not uncover any serious shortcomings in the design. Below is a list — in no particular order — over the areas where major improvements can readily be made to the ACT:

1. A more expressive content language for the knowledge base$_{5.3}$. Many possibilities exist, but a careful choice need to be made.
2. Planning capabilities for the ACMEs. One might wonder how hard a problem this will be$_{21.2}$.
3. A more sophisticated message protocol, possibly based on standardised protcols like the ones described in [14].
4. Code optimisations in the ACC. A few cases are downright obvious (the present ACC always produces a `for`-loop when compiling an `IF`-statement, for example), but countless possibilities exist.
5. A more sophisticated strategy finder$_{18.5.2,18.5.1}$. This might be based on the work carried out by the Diplomacy Programming Project [5].

Chapter 22

Conclusion

This thesis describes the the implementation of a new agent programming language, AgentC, and a software toolkit (ACT) which complements the AgentC language with a compiler (ACC) and components from which to build agents. The ACT additionally provides a substantial standardised test bed (the Haplomacy game), which provides a benchmark application for agent-based systems.

Many APLs of various kinds have been suggested over the years; fewer have actually been implemented, though, and even fewer have shown good results in practical applications. The same is true of toolkits from which to build agents.

AgentC belongs to the exclusive club of APLs which actually are *computable*, which provide a working implementation and which are also relatively efficient when it comes to computational complexity (the simplicity of the langauge certainly helps in this regard). No part of the AgentC can be said to be groundbreaking, though, since the ACC uses only very traditional techniques.

The most controversial part of the ACT is arguably the design strategy, although this strategy is not uncommon outside of agent-related areas: The ACT is constructed *bottom-up*, seamlessly integrating a new layer of agent-oriented functionality to traditional, tried-and-tested object-oriented programs. The reader need but to consult a survery of APLs, such as e.g. [22], to be convinced that it is quite common to apply a top-down agent-based approach to even complex problems, even though "*. . . there is no evidence that any system developed using agent technology could not have been built just as easily using non-agent techniques*" [38].

Like any other APL (or toolkit for that matter) the AgentC (or ACT) reportedly$_{20}$ have shown good results in a certain application. Whether or not the AgentC (or ACT) is *useful*, however, is an entirely different matter. However, the bottom-up approach, the use of (at places) simple but also quite generic components, and the implementation of the Haplomacy test bed application, makes the ACT quite well suited as the basis for experiments with agent-based systems; the question of whether the ACT is useful to conduct such experiments will hopefully be answered in the future by a DTU student.

Part VI

Appendices

Appendix A

AgentC Grammar

A.1 Lexical Syntax

$$
\begin{array}{lcl}
\langle\text{Identifier}\rangle & ::= & \langle\text{IdentifierStart}\rangle\ \big(\langle\text{Letter}\rangle\ \big|\ \langle\text{Digit}\rangle\big)^{*} \\
\langle\text{IdentifierStart}\rangle & ::= & \texttt{a}\ \ldots\ \texttt{z}\ \big|\ \texttt{\$}\ \langle\text{Letter}\rangle \\
\langle\text{Letter}\rangle & ::= & \texttt{A}\ \ldots\ \texttt{Z}\ \big|\ \texttt{a}\ \ldots\ \texttt{z}\ \big|\ \texttt{\_} \\
\langle\text{Digit}\rangle & ::= & \texttt{0}\ \big|\ \langle\text{NonZeroDigit}\rangle \\
\langle\text{NonZeroDigit}\rangle & ::= & \texttt{1}\ \ldots\ \texttt{9} \\
\langle\text{Variable}\rangle & ::= & \texttt{?}\ \langle\text{Letter}\rangle\ \big(\langle\text{Letter}\rangle\ \big|\ \langle\text{Digit}\rangle\big)^{*}\ \big|\ \texttt{\_} \\
\langle\text{AttitudeToken}\rangle & ::= & \texttt{\#}\ \langle\text{Letter}\rangle\ \big(\langle\text{Letter}\rangle\ \big|\ \langle\text{Digit}\rangle\big)^{*} \\
\langle\text{Literal}\rangle & ::= & \langle\text{IntegerLiteral}\rangle\ \big|\ \langle\text{DoubleLiteral}\rangle\ \big|\ \langle\text{StringLiteral}\rangle \\
\langle\text{IntegerLiteral}\rangle & ::= & \texttt{0}\ \big|\ \texttt{-}^{?}\ \langle\text{NonZeroDigit}\rangle\ \langle\text{Digit}\rangle^{*} \\
\langle\text{DoubleLiteral}\rangle & ::= & \langle\text{Digit}\rangle^{+}\ \texttt{.}\ \ \langle\text{Digit}\rangle^{*}\ \langle\text{Exponent}\rangle^{?} \\
 & & \big|\ \ \texttt{.}\ \ \langle\text{Digit}\rangle^{+}\ \langle\text{Exponent}\rangle^{?} \\
 & & \big|\ \langle\text{Digit}\rangle^{+}\ \langle\text{Exponent}\rangle \\
\langle\text{Exponent}\rangle & ::= & \big(\texttt{e}\ \big|\ \texttt{E}\big)\ \big(\texttt{+}\ \big|\ \texttt{-}\big)^{?}\ \langle\text{Digit}\rangle^{+} \\
\langle\text{StringLiteral}\rangle & ::= & \textit{Any}\ \text{Java}\ \textit{string literal}
\end{array}
$$

A.2 Program Structure

```
⟨Program⟩              ::=  ⟨CompilationModule⟩*
⟨CompilationModule⟩    ::=  ⟨Definitions⟩ | ⟨Attitudes⟩ |
                            ⟨Facts⟩ | ⟨Procedure⟩
⟨Definitions⟩          ::=  DEFS { ⟨Definition⟩* }
⟨Definition⟩           ::=  ⟨Identifier⟩ = ⟨Literal⟩ ;
⟨Attitudes⟩            ::=  ⟨AttitudeDecl⟩*
⟨AttitudeDecl⟩         ::=  ⟨AttitudeToken⟩ = ⟨IntegerLiteral⟩ ;
⟨Facts⟩                ::=  ⟨InitialFact⟩*
⟨InitialFact⟩          ::=  ⟨Fact⟩ ;
⟨Procedure⟩            ::=  PROCEDURE ⟨Identifier⟩ ⟨ParameterList⟩
                            (; | ⟨ProcedureBody⟩)
⟨ParameterList⟩        ::=  ( (⟨Variable⟩ (, ⟨Variable⟩)*)? )
⟨ProcedureBody⟩        ::=  { ⟨Rule⟩* }
```

A.3 Values

```
⟨Value⟩              ::=  ⟨Expression⟩ | ⟨Sentence⟩
⟨Expression⟩         ::=  ⟨Literal⟩ | ⟨Variable⟩ | ⟨SymbolReference⟩ |
                          ⟨SelfReference⟩ | ⟨ProcedureCall⟩
⟨SymbolReference⟩    ::=  ⟨Identifier⟩
⟨SelfReference⟩      ::=  SELF
⟨ProcedureCall⟩      ::=  CALL ⟨identifier⟩ ⟨TermList⟩
⟨Query⟩              ::=  Q ⟨Identifier⟩ ⟨TermList⟩
⟨TermList⟩           ::=  ( (⟨Expression⟩ (, ⟨Expression⟩)*)? )
⟨Sentence⟩           ::=  ⟨Fact⟩
⟨Fact⟩               ::=  ⟨AttitudeToken⟩ ⟨Identifier⟩ ⟨TermList⟩
```

A.4 Rules

```
⟨Rule⟩                ::=  ⟨Statement⟩ | ⟨MessageRule⟩
⟨Statement⟩           ::=  ⟨Action⟩ | ⟨Assignment⟩ | ⟨IfStatement⟩ |
                           ⟨LetStatement⟩ | ⟨LockedStatement⟩ |
                           ⟨MentalUpdate⟩ | ⟨ReturnStatement⟩ |
                           ⟨SayStatement⟩
⟨Block⟩               ::=  { ⟨Statement⟩* }
⟨ActionStatement⟩     ::=  ⟨DoStatement⟩ | ⟨XeqStatement⟩
⟨Assigmnent⟩          ::=  ⟨Variable⟩ = ⟨Expression⟩ ;
⟨DoStatement⟩         ::=  DO ⟨Identifier⟩ ⟨TermList⟩ ;
⟨XeqStatement⟩        ::=  XEQ ⟨Identifier⟩ ⟨TermList⟩ ;
⟨ReturnStatement⟩     ::=  RETURN ⟨Expression⟩? ;
⟨IfStatement⟩         ::=  IF ⟨Guard⟩ ⟨Block⟩
                           (ELSIF ⟨Guard⟩ ⟨Block⟩)*
                           (ELSE ⟨Block⟩)?
⟨Guard⟩               ::=  ( (⟨Condition⟩ (, ⟨Condition⟩)*)? )
⟨Condition⟩           ::=  ⟨Comparison⟩ | ⟨NegatedCondition⟩ |
                           ⟨MentalCondition⟩ | ⟨Action⟩ | ( ⟨Condition⟩ )
⟨Comparison⟩          ::=  ⟨Expression⟩ ⟨Relation⟩ ⟨Expression⟩
⟨Relation⟩            ::=  < | <= | = | != | >= | >
⟨NegatedCondition⟩    ::=  ! ⟨Condition⟩
⟨MentalCondition⟩     ::=  ⟨Fact⟩ (AS ⟨Variable⟩)?
⟨LetStatement⟩        ::=  LET ⟨Variable⟩ = ⟨Expression⟩ ;
⟨LockedStatement⟩     ::=  LOCKED ⟨Block⟩
⟨MentalUpdate⟩        ::=  ⟨AdoptStatement⟩ | ⟨DropStatement⟩
⟨AdoptStatement⟩      ::=  ADOPT (⟨Fact⟩ | ⟨Variable⟩) ;
⟨DropStatement⟩       ::=  DROP (⟨Fact⟩ | ⟨Variable⟩) ;
⟨SayStatement⟩        ::=  SAY ⟨MessagePattern⟩ ;
⟨MessagePattern⟩      ::=  [ (⟨Attribute⟩ (, ⟨Attribute⟩)* )? ]
⟨Attribute⟩           ::=  ⟨Identifier⟩ = ⟨Value⟩
⟨MessageRule⟩         ::=  WHEN (NOTHING | ⟨MessageGuard⟩) ⟨Block⟩
⟨MessageGuard⟩        ::=  [ (⟨xAttribute⟩ (, ⟨xAttribute⟩)* )? ]
⟨xAttribute⟩          ::=  ⟨Identifier⟩ = (⟨Expression⟩ |
                           ⟨Fact⟩ (AS ⟨Variable⟩)?)
```

Appendix B

Adjudicating Orders in Haplomacy

Orders are adjudicated according to two general principles: a set of *rules* which prescribe what will happen in a given case, and an *adjudication principle* which prescribes how to apply the rules.

The rules are described in appendices B.2 and B.2.1, while the adjudication principle is described in appendix B.3.

B.1 Example Diagrams

The diagrams shown in the following correspond to the relevant diagrams in [4]. Each diagram consists of two parts, the first of which shows an example scenario. Movement orders are shown as large arrows pointed towards the destination of the move, while support orders are shown as small arrows directed at the supported unit; the arrows bear the colour of the supported unit.

The second part shows how the situation is to be resolved. The diagrams are actual screenshots[1] from the test bed application, and the results shown

[1] The sections correspond to the centre of the default game board (D3–F5), where the indication of E4 as a support centre has been removed.

have been obtained by applying the implemented adjudicator on the scenario[2].

In the diagram texts all references to provinces and units use the number conventions shown in figure B.1 below; a single capital '*P*' is prefixed to the number to indicate a province, while '*U*' is used as a prefix to indicate a unit.

0	1	2
3	4	5
6	7	8

Figure B.1: Reference number conventions for the examples.

B.2 HAPLOMACY Rules

§1. *All units have the same strength.*

§2. *There can be only one unit in a province at a time.*

§3. *Equal strength units trying to occupy the same province cause all those units to remain in their original provinces.*
See figure B.2 on page 181.

§4. *A standoff does not dislodge a unit already in the province where the standoff took place.*
See figure B.2 on page 181.

§5. *One unit not moving can stop a series of other units from moving.*
See figure B.3 on page 181.

§6. *Units cannot trade places [without the use of a convoy].*
See figure B.4 on page 181.

§7a. *Three or more units can* **not** *rotate provinces during a turn [provided none directly trade places].*
This rule has been altered for two reasons:

[2] In addition to being illustrative, therefore, the diagrams also serve to verify correct operation of the implementation.

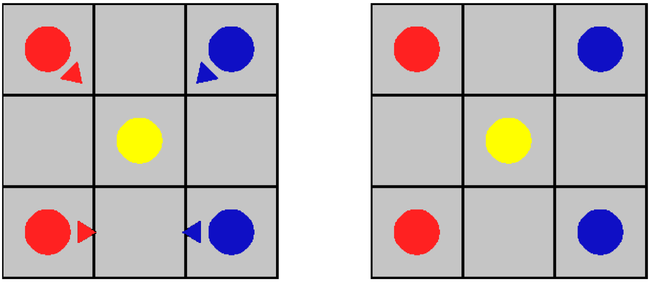

Figure B.2: $U0$ and $U2$ both try to enter $P4$, causing a standoff (but without dislodging $U4$). Similarly, $U6$ and $U8$ cause a standoff by simultaneously trying to enter $P7$.

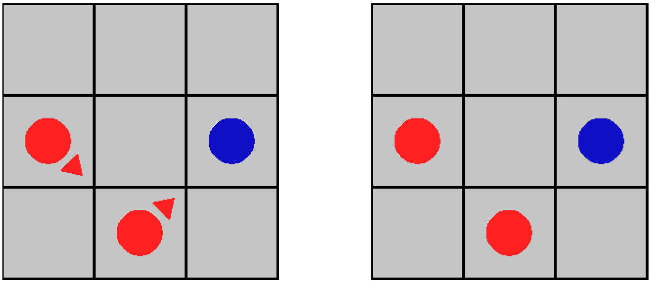

Figure B.3: Because $U7$ cannot move, neither can $U3$

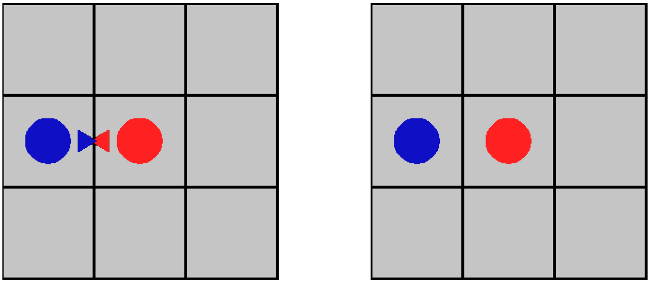

Figure B.4: Units cannot trade places.

- The implementation is simplified (no detection of cycles is necessary).
- The rule in [4] is ambiguous[3].

See figure B.5 on page 182.

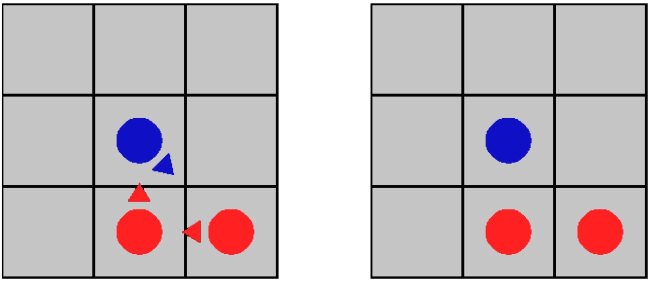

Figure B.5: Units cannot trade places, even when three or more units form a cycle.

§8a. If a holding unit is attacked by a larger army (a unit which has more supports) the unit will be dislodged.
See figure B.6 on page 182.

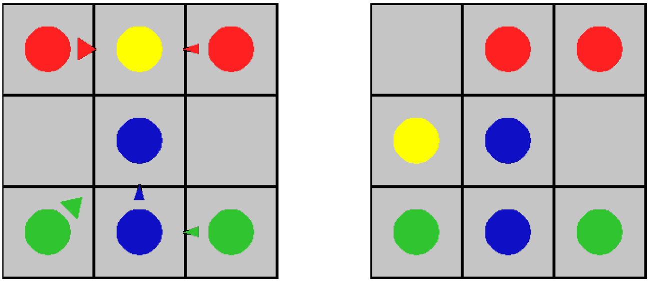

Figure B.6: Because $U0$ has support (it is an army of size 2), it dislodges $U1$, which is forced to retreat. Since $U4$ and $U6$ are armies of equal size, $U6$ cannot dislodge $U4$.

[3] For example, it is not specified whether the cycle should be allowed to move or not, in case one or more of the provinces in the cycle are under attack by a single unit

§9a. If no single largest army is attacking or defending a province, the result will be a standoff — even if a smaller army is present in that province.
See figure B.7 on page 183.

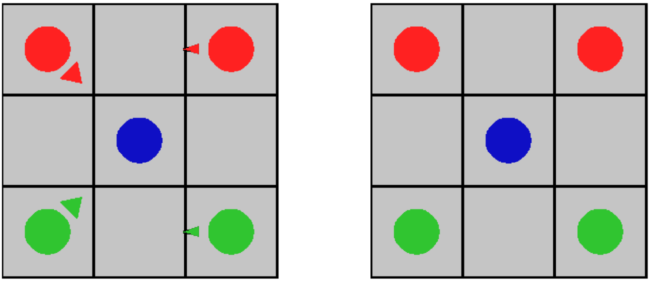

Figure B.7: Even though $U0$ and $U6$ are larger armies than $U4$, the resulting standoff will not cause $U4$ to be dislodged.

§10. *A dislodged unit can still cause a standoff in a province different from the one that dislodged it.*
See figure B.8 on page 183.

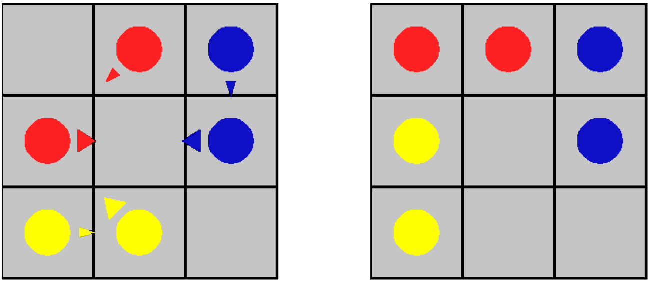

Figure B.8: Even though the supported attack from $P7$ dislodges $U3$ (forcing it to retreat to $P0$), the unit still causes a standoff with $U5$ in $P4$.

§11. *A dislodged unit, even with support, has no effect on the province that dislodged it.*
See figures B.9 on page 184 and B.10 on page 184.

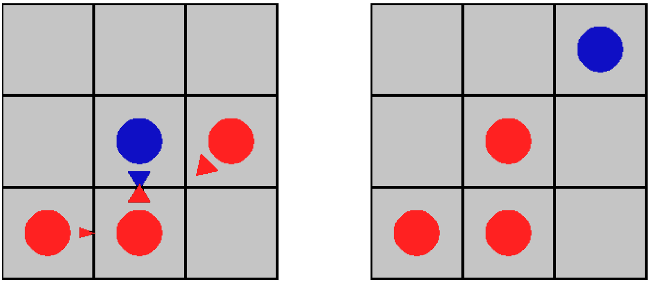

Figure B.9: Because $U4$ is dislodged by an attack from $P7$, $U4$ cannot prevent $U5$ from moving into $P7$.

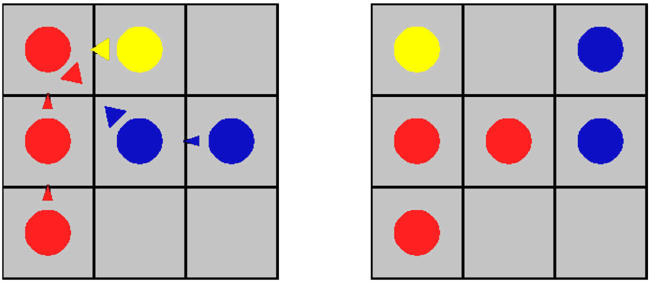

Figure B.10: Even though $U4$ is a larger army than $U1$ it cannot prevent $U1$ from occupying $P0$, because $U4$ was dislodged by an attack from that province.

§12. *A country cannot dislodge or support the dislodgment of one of its own units, even if that dislodgement is unexpected.*
See figure B.11 on page 185, figure B.13 on page 186 and figure B.12 on page 185.

§13. *Support is cut if the unit giving support is attacked from any province except the one where support is being given.*
See figures B.14 on page 186 and B.15 on page 187.

§14. *Support is cut if the supporting unit is dislodged.*
See figure B.16 on page 187.

§15. *A unit being dislodged by one province can still cut support in another.*
See figure B.17 on page 187.

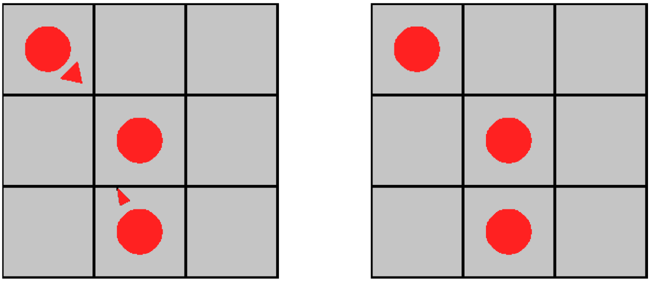

Figure B.11: The supported attack by $U0$ on $U4$ cannot dislodge $U4$, because $U0$ and $U4$ belong to the same player

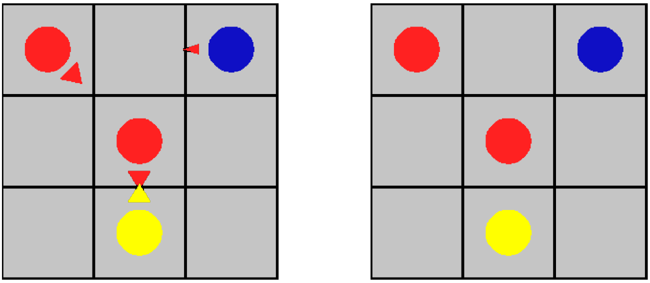

Figure B.12: Because neither $U7$ nor $U4$ can move, $U0$ effectively attacks and dislodges $U4$ with the support of $U2$. Since $U0$ and $U4$ belong to the same player, however, the attack fails.

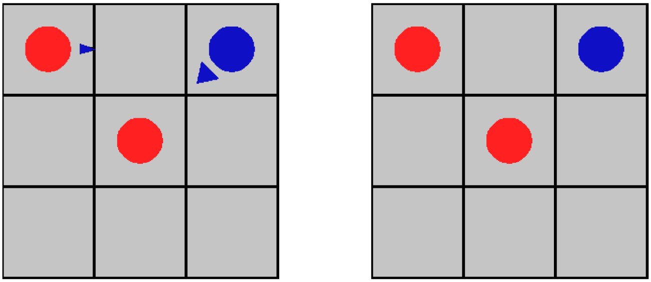

Figure B.13: Even though $U0$ supports $U2$, which normally would cause $U4$ to be dislodged, the move fails because $U0$ and $U2$ belong to the same player.

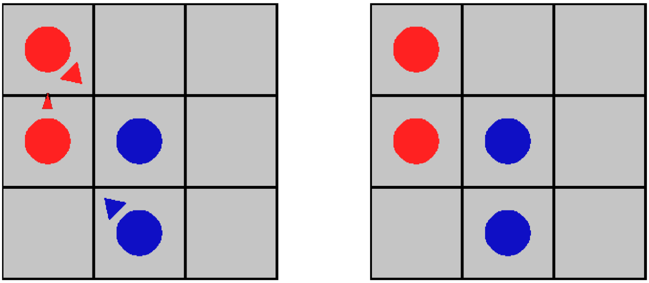

Figure B.14: Because $U7$ attacks $U4$ its support is cut. Hence $U0$ must stand in its place because it cannot dislodge $U4$ on its own.

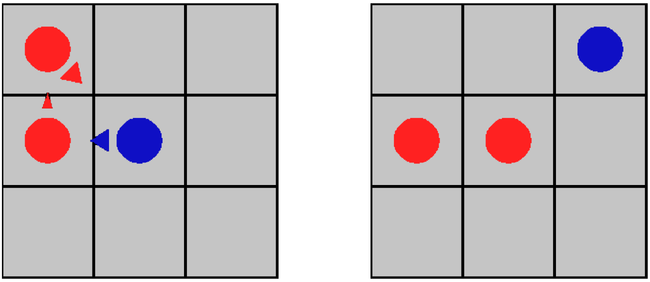

Figure B.15: Because $U3$ is giving support into $P4$ the attack by $U4$ does not cut support. Hence, $U4$ is dislodged and forced to retreat.

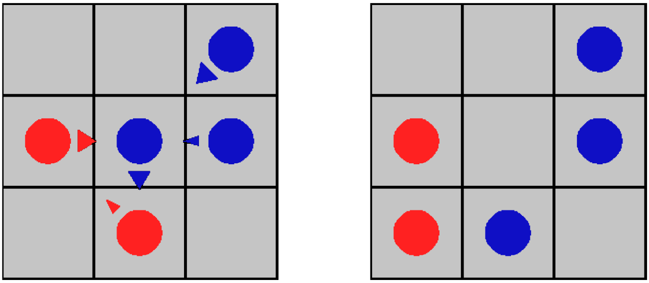

Figure B.16: Because the supported attack by $U4$ dislodges $U7$, its support is cut. The effect is a standoff by $U3$ and $U2$ in $P4$.

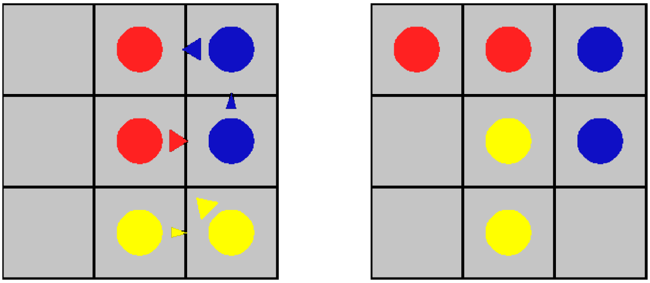

Figure B.17: Although $U4$ is dislodged by the supported attack by $U8$, it still manages to cut support from $U5$. The result is that $U2$ cannot move into $P1$

§16. *An attack by a country on one of its own units does not cut support.* See figure B.18 on page 188.

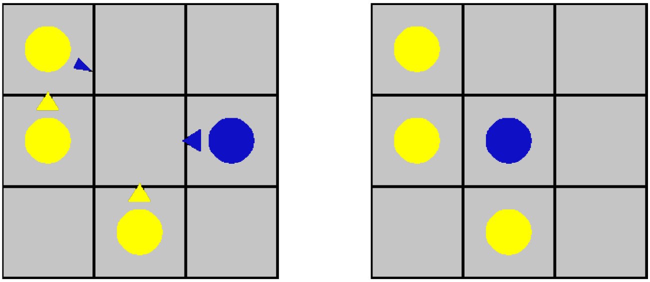

Figure B.18: Because $U0$ and $U3$ are owned by the same player the attack by $U3$ on $U0$ does not cut support. This support allows $U5$ to enter $P4$, forcing $U6$ to stand in its place.

B.2.1 About the Rules

The rules of HAPLOMACY have been modelled as closely as possible on the rules of Diplomacy [4], with a few exceptions dictated by practical issues$_{B.3}$, and a series of simplifications due to the generalised game board and the absence of fleets.

The rules in the following are quoted verbatim from page 23 of [4], with the only modification that amendments are shown in **bold face**. The rules are numbered as the corresponding rules in [4][4]

B.3 Adjudication Principles

The rules described in the preceding$_{B.2}$ are generally not sufficient to resolve an arbitrary set of orders: more than one of the rules may apply, and the

[4] Except for the replacement of rules 8 and 9 and the amendment to rule 7, the rules of HAPLOMACY directly correspond to the first 16 of the 22 rules on page 23 of [4]. The remaining 6 rules all relate to the *convoy* order (for fleet units), and are hence not relevant to HAPLOMACY

outcome is dependent on which rule is applied first. Oddly enough, the rules described in [4] — although they are numbered — do not constitute an *algorithm*: there is no fixed way to determine how a complex situation should be analysed in order to decompose it into the simple situations for which rules have been defined. One general guideline is given:

> *"In complicated situations, it helps to first determine what support, if any, is cut. Once this is determined, it is easier to resolve orders."*

While the guideline gives a helpful hint on how an adjudication algorithm could be constructed, it does not suffice even in simple cases: consider the example shown in figure B.16 on page 187. The outcome depends on support being cut for a dislodged unit, *but if support are to be cut first, it may not yet have been determined whether a unit should be dislodged*!

Because a HAPLOMACY *adjudicator* (a component which is responsible for adjudicating the outcome of a set of orders) needs to be implemented in software, it is paramount that a detailed algorithm exists. Not only will such an algorithm be necessary to ensure that the outcome of any set of orders is known[5], the algorithm is essential when it comes to making the program work: if the program does not take into account *any* possible situation, then the program will fail once one of the unforeseen situations occur[6].

The conclusion of the above is that [4] does not define a usable adjudication algorithm, hence one must be designed from scratch. The design must take several demands into account:

- The algorithm must handle every possible situation in a reasonable way.
- The algorithm should be reasonably efficient in terms of computation complexity.

[5] In the Diplomacy rules [4], according to the tournament rules [6], "*There are two known Paradoxes which are rather rare and should they occur then the moving units hold*". What these paradoxes should be taken to mean is not explained further, however.

The algorithm presented here is *not entirely deterministic* because an arbitrary choice is made at some points. However, this choice is only made as a last resort.

[6] It may be so that the rules in [4] — with the aforementioned amendment in [6] — are sufficient to adjudicate any Diplomacy game occurring in practice in a deterministic way. However, because HAPLOMACY is a generic game the probability of a single legal combination occurring can not be disregarded — and when it comes to writing a working program, *no* probability larger than 0 can be disregarded.

- The algorithm should preferably produce the same results as shown in the examples of [4].

The two of the first demands can be relatively easily met: an efficient algorithm which handles every possible situation can be obtained merely by enforcing strict restrictions on which orders will succeed. However, it is desirable that Haplomacy resembles Diplomacy to a high degree: The popularity of Diplomacy has not declined with time; there is no question, therefore, that the game itself strikes a good balance between complexity and playability — properties which are equally desirable in Haplomacy.

The resulting algorithm has therefore been designed in such a way that *every applicable rule of* Diplomacy *should be included in* Haplomacy, with the only exception that rule 7 has been negated to forbid the movement of units in circles — the added complexity of detecting that situation, the relatively low probability at which it occurs, and the fact that [4] does not elaborate on the situation makes it an obvious choice for omission.

Even though the applicable rules of Diplomacy also apply in Haplomacy, there is generally no guarantee that the resulting adjudicator will act entirely as would be expected by a seasoned Diplomacy player. In the example cases used in [4] — which have led to the example cases shown in appendix B.2 — the results happen to coincide, however. The algorithm presented below therefore ensures that the two games are adjudicated similarly, at least in the simple cases.

B.3.1 The Adjudication Algorithm

The algorithm depends on the data structure described in chapter 18, and the terminology defined there will used in the following.

Let the *target* of an army (which consists of any number of units ordered to support one single unit, plus that single unit itself) be defined as follows:

- If an army leader is ordered to move, then the target is the province to which that unit is directed.
- Otherwise, the target is the location of the army leader (by definition, then, this unit will be ordered to hold).

In either case the target is said to be *targeted* by the supported unit. A province which is targeted by at least one army is said to be *under attack*.

The conjunction of all armies targeting a single province is collectively known as a *dispute* over that target.

The adjudication algorithm is designed to resolve orders iteratively rather than all at once. The benefit is to reduce the complexity of each iteration step. In order to reduce memory consumption, the algorithm allows the game board state to be modified directly in each step. The overall structure of the algorithm is therefore:

1. Cancel all invalid orders.
2. Cut support subject to rules 13 and 16, *but not to rule 14*.
3. As long as there are units not ordered to hold
   (a) Identify the dispute to be dealt with next.
   (b) Handle the selected dispute.
   (c) Repeat.

In any step of the algorithm, the next dispute to be handled is found in the following way:

- If a vacant province which is targeted by only one army exists, then select that province[7]
- Otherwise, select the dispute involving the largest army on the game board[8].
  - In case more than one dispute is eligible, prefer a dispute in which the targeted province is occupied by a unit which is ordered to support[9].
  - In case more than one eligible dispute remains, and if the largest army is a single unit, then prefer the dispute involving the largest number of units[10].

[7] The situation is resolved first because the dispute has no effect on the outcome of any other disputes.

[8] Selecting the largest army will make that army succeed in its enterprise (move or hold), which is generally what is intended. However, some instances where rule 14 would apply are not detected: if one of its supports is dislodged by an army whose size is only one less, then the army is not actually the largest army on the game board; but that army may itself have one of its support dislodged by another army, etc.

By always choosing the largest army first in each step, however, it will be guaranteed that the algorithm always terminates.

[9] This special case is responsible for ensuring that rule 14 is used in some situations, such as figure B.16 on page 187 — but not that the rule is always applied

[10] Handling the most units in each step is generally a good way of ensuring algorithm performance. If the largest army does not have support, no units on the game board are giving support (since the largest army is eliminated in every step of the algorithm), and

- ○ Otherwise, make an arbitrary choice between the eligible disputes[11].

In the following, *carrying out* an order for a unit means that

1. If the unit is moving, then move the unit to its destination.
2. Cancel the unit's orders

where *cancelling* an order means

1. Order the unit to hold.
2. Order every other unit which supports the unit to hold.

Finally, the occupant of a province which is under dispute is denoted the *defender* of that dispute. With these definitions at hand, the meaning of *handle the selected dispute* in the preceding algorithm description can be specified as follows:

- If the dispute involves precisely one army which is larger than any other army involved, and if the leader of that army is moving, then the largest army is declared the *winning army.* The following possible cases exist:
    1. If the winning army leader is owned by the player to which the defender (if any) belongs, or if the winning army leader is supported by said player, then handle the dispute as a draw (c.f. rule 12).
    2. Otherwise, if a defender exists, dislodge that unit from the targeted province, and add it onto a list of retreats to be resolved later. Then there is a choice:

therefore it is safe to choose the dispute involving the largest number of units without affecting the outcome of other disputes — at this time any remaining unit on the game board will be part of a standoff (because vacant provinces under attack by only one army already have been handled). But in order to detect these standoffs — which is necessary in order to correctly resolve subsequent retreats, and which is carried out as a side-effect of the adjudication — the algorithm must be allowed to run once for every dispute; otherwise this case could have been more efficiently handled.

[11] Because HAPLOMACY is a generic game allowing any size of game board, there is no fixed limit on the number of eligible disputes at this point. Therefore, there is theoretically no limit to the number of criterions which necessary to one single dispute in a deterministic way. The algorithm will therefore necessarily involve an arbitrary choice at some point, although the probability of that choice occurring is smaller when more criteria are being employed. Since the algorithm presented here is sufficient to duplicate the examples of [4], it has been decided to make the arbitrary choice at this point.

        - If the defender is moving into the location of the winning army leader or is supporting a move into that province, then cancel the defender's orders (c.f. rule 11).
        - Otherwise, leave the defender's orders to be resolved later (c.f. rule 10).
    3. Move the winning army leader into the targeted province.
    4. Cancel the orders of any other unit involved in the dispute.
- Otherwise, the result is a *draw*, which effectively is a standoff between the involved armies (although it may come about as the result of the defender being the strongest army). The following steps are then necessary:
    1. Record that the targeted province was the scene of a standoff (this information is necessary to make the subsequent retreats).
    2. If there exists a defender which bas been ordered to hold, then cancel the orders of the defender (it is either in standoff or successfully holding its location).
    3. If there exists a defender which has been ordered to move into the province from where the largest army came, then cancel the orders of the defender (it is in standoff with the largest army).
    4. Cancel the orders of any other unit involved in the dispute.

Appendix C

Sample AgentC Code

This appendix lists the AgentC source code for four different ACMEs which can have been used to play Haplomacy in the setup described in chapter 19; each ACMEs exhibits a unique behaviour, or 'personality', from which their respective names originate. The comments in the code describe how this 'personality' manifests itself.

C.1 Common Code

The code below is shared by all the different ACMEs; it has been placed in a separate file for the same reason.

```
// these attitudes are used by default
ATTITUDES {
    #DID = 0;
    #BELIEVE = 1;
    #B = 1;
    #INTEND = 2;
    #I = 2;
}


// the initial relations to all players ($INITIAL_RELATION must
// be specified elsewhere)
FACTS {
    #B relation(SELF, $RED, $INITIAL_RELATION);
    #B relation(SELF, $BLUE, $INITIAL_RELATION);
    #B relation(SELF, $GREEN, $INITIAL_RELATION);
    #B relation(SELF, $YELLOW, $INITIAL_RELATION);
}


// Exclude beliefs about the agent's relations with itself.
// The simplicity of the FACTS module prevents this case from being
// detected
PROCEDURE init() {
    DROP #B relation(SELF, SELF, _);
}


// definitions for the utility procedures
DEFS {
    $NEGONE = -1.0;
    $NEGHALF = -0.5;
    $NEGQUART = -0.25;
    $ZERO = 0.0;
    $QUART = 0.25;
    $HALF = 0.5;
    $ONE = 1.0;
}


// adjust the relation to a given player by a certain relative amount,
// bounded by $NEGONE and $ONE
PROCEDURE adjustRelation(?player, ?delta) {
    LOCKED {
        IF (#B relation(SELF, ?player, ?X) AS ?old) {
            DROP ?old;
            LET ?y = Q add(?X, ?delta);
            IF (?y > $ONE) {
                ADOPT #B relation(SELF, ?player, $ONE);
                RETURN $ONE;
            }
            ELSIF (?y < $NEGONE) {
                ADOPT #B relation(SELF, ?player, $NEGONE);
                RETURN $NEGONE;
            }
            ELSE {
                ADOPT #B relation(SELF, ?player, ?y);
                RETURN ?y;
```

```
         }
      }
   }
}


// ensure that the given relation at least has the specified value
PROCEDURE ensureMinimumRelation(?player, ?value) {
   LOCKED {
      IF (#B relation(SELF, ?player, ?r) AS ?old) {
         IF (?r < ?value) {
            DROP ?old;
            ADOPT #B relation(SELF, ?player, ?value);
         }
      }
   }
}


// ensure that the given relation at most has the specified value
PROCEDURE ensureMaximumRelation(?player, ?value) {
   LOCKED {
      IF (#B relation(SELF, ?player, ?r) AS ?old) {
         IF (?r > ?value) {
            DROP ?old;
            ADOPT #B relation(SELF, ?player, ?value);
         }
      }
   }
}



// Handle negotiations in three separate procedures
PROCEDURE negotiate() {
   WHEN NOTHING {
      // make a new inquiry
      RETURN CALL inquire();
   }

   WHEN [contents=$ACCEPT] {
      // the message is a reply
      RETURN CALL handleReply();
   }
   WHEN [contents=$REJECT] {
      // the message is a reply
      RETURN CALL handleReply();
   }

   // else the message is a notification or an inquiry
   RETURN CALL handleNewMessage();
}


// the inquire() procedure produces new inquiries.
PROCEDURE inquire();

// the handleReply() procedure should handle a reply to the last inquiry
```

```
PROCEDURE handleReply();

// the handleNewMessage() should handle a new notification or inquiry
PROCEDURE handleNewMessage();
```

C.2 The Ruthless Player

```
/**
 * A ruthless player.
 */
DEFS {
    // the initial relation to all players
    $INITIAL_RELATION = 0.25;
    $RANDOM_RANGE = 0.15;
}


PROCEDURE inquire() {
    // the ruthless player doesn't bother about making conversations.
    // however, a slight randomization is added to the player's relations
    IF (#DID updateRelations()) {
        RETURN;
    }

    IF (#B relation(SELF, ?P, ?r)) {
        LET ?R = Q random($RANDOM_RANGE);
    }


    ADOPT #DID updateRelations();
}


PROCEDURE handleReply() {
    // the replies are insignificant,
}


PROCEDURE handleNewMessage() {
    // handle a request for switching alliance
    WHEN [type=$REQUEST, from=?P, contents=#I relation(?p, ?l, ?u) AS ?c] {
        IF (Q strengthOf(?P) > Q strengthOf(SELF)) {
            IF (Q strengthOf(?p) < Q strengthOf(SELF)) {
                // use the lowest value for a weaker player
                CALL ensureMinimumRelation(?p, ?l);
            }
            ELSE {
                // use the highest value, bounded by $HALF, for
                // a stronger player
                IF (?u > $HALF) {
                    CALL ensureMinimumRelation(?p, $HALF);
                }
                ELSE {
                    CALL adjustRelation(?p, ?u);
                }
            }
            SAY [to=?P, type=$ACCEPT, contents=?c];
        }
        ELSE {
            SAY [to=?P, type=$REJECT, contents=?c];
        }
    }
}
```

```
PROCEDURE giveOrders() {
    // attack the player with the lowest relation, regardless of the absolute value.
    // otherwise stick to the relations.
    XEQ resetFriends();
    LET ?minRelation = $ONE;
    LET ?minPlayer = -1;

    IF (#B relation(SELF, ?P, ?R)) {
        IF (?R < ?minRelation) {
            ?minRelation = ?R;
            ?minPlayer = ?P;
        }
        IF (Q strengthOf(?P) > Q strengthOf(SELF), ?R > $ZERO) {
            XEQ registerFriend(?P);
        }
    }

    XEQ unregisterFriend(?minPlayer);


    IF (!XEQ attack(Q add(0.50, Q random(0.2)))) {
        XEQ defend();
    }
}


PROCEDURE updateStatus() {
    WHEN NOTHING {
        // get ready for next turn
        DROP #DID updateRelations();
    }
    WHEN [type=$PLAYER_ELIMINATED, contents=?X] {
        DROP #B relation(?X, _);
        RETURN;
    }
    WHEN [type=$UNIT_ATTACKED, contents=?X] {
        CALL adjustRelation(?X, $NEGHALF);
        RETURN;
    }
    WHEN [type=$SUPPORT_CENTRE_ATTACKED, contents=?X] {
        CALL adjustRelation(?X, -0.75);
        RETURN;
    }
    WHEN [type=$SUPPORT_CENTRE_CONQUERED, contents=?X] {
        CALL adjustRelation(?X, $NEGONE);
        RETURN;
    }
}
```

C.3 The Vindictive Player

```
/**
 * The vindictive player.
 */
DEFS {
   // the initial relation to all players
   $INITIAL_RELATION = 0.75;
}


PROCEDURE inquire() {
   // all inquiries have been sent for this turn - don't make further conversations
   IF (#DID sendMessages()) {
      RETURN;
   }

   // Send a petition to all friendly players to attack the enemies
   IF (#B relation(SELF, ?E, ?er), ?er < $ZERO,
         #B relation(SELF, ?F, ?fr), ?fr > $ZERO) {
      SAY [to = ?F, type = $REQUEST,
            contents = #I relation(?E, $NEGONE, $ZERO)];
   }

   // don't make further conversations
   ADOPT #DID sendMessages();
}


PROCEDURE handleReply() {
   // don't care about replies - they don't affect the behaviour.
}

PROCEDURE handleNewMessage() {
   // handle a request for switching alliance
   WHEN [type=$REQUEST, from=?P, contents=#I relation(?p, ?l, ?u) AS ?c] {
      IF (#B relation(SELF, ?P, ?r), ?r >= $ZERO) {
         // the sender is friendly: try to honour the request
         IF (#B relation(SELF, ?p, ?pr) AS ?o) {
            IF (?pr <= $ZERO) {
               IF (?l > $ZERO) {
                  // can't forgive enemy ?p : reject
                  SAY [type=$REJECT, to=?P, contents=?c];
               }
               ELSE {
                  // accept the lower bound for the
                  // relation (it is not larger than 0.0)
                  DROP ?o;
                  ADOPT #B relation(SELF, ?p, ?l);
                  SAY [type=$ACCEPT, to=?P, contents=?c];
               }
            }
            ELSIF (?l > 0.0) {
               // accept the lower bound if the player
               // relation is friendly
               CALL ensureMinimumRelation(?p, ?l);
               SAY [type=$ACCEPT, to=?P, contents=?c];
            }
```

```
                ELSE {
                    // don't make enemies just because it is
                    // suggested
                    SAY [type=$REJECT, to=?P, contents=?c];
                }
            }
        }
        ELSE {
            // reject all enemies
            SAY [type=$REJECT, to=?P, contents=?c];
        }
        RETURN;
    }
}


PROCEDURE giveOrders() {
    // calculate a new set of friendly players
    XEQ resetFriends();
    IF (#B relation(SELF, ?P, ?R), ?R >= $ZERO) {
        IF (#DID attack(?R)) {
            // ?P has previously attacked a support centre -
            // this is not forgotten: consider the player
            // friendly only half of the time, even if relations
            // are good
            IF (Q random() < $HALF) {
                XEQ registerFriend(?P);
            }
        }
        ELSE {
            // ?P has never attacked support centres -
            // consider ?P a loyal friend.
            XEQ registerFriend(?P);
        }
    }
    // else the player is an enemy


    IF (!XEQ attack(Q add(0.60, Q random(0.15)))) {
        XEQ defend();
    }
}


PROCEDURE updateStatus() {
    WHEN NOTHING {
        // get ready for the next round
        DROP #DID sendMessages();
        RETURN;
    }
    WHEN [type=$PLAYER_ELIMINATED, contents=?X] {
        DROP #B relation(?X, _);
        RETURN;
    }
    WHEN [type=$UNIT_ATTACKED, contents=?X] {
        CALL adjustRelation(?X, $NEGHALF);
        RETURN;
    }
    WHEN [type=$SUPPORT_CENTRE_ATTACKED, contents=?X] {
        CALL adjustRelation(?X, -0.75);
        ADOPT #DID attack(?X);
        RETURN;
    }
    WHEN [type=$SUPPORT_CENTRE_CONQUERED, contents=?X] {
        // subtract 2.0 from the relation: this means WAR!
        CALL adjustRelation(?X, -2.0);
        ADOPT #DID attack(?X);
        RETURN;
    }
}
```

C.4 The Cautious Player

```
/**
 * The cautious player.
 */
DEFS {
    // the initial relation to all players
    $INITIAL_RELATION = 1.0;

    $EIGHTH = 0.125;
}


PROCEDURE inquire() {
    IF (#DID finishInquiries()) {
        RETURN;
    }

    IF (#B relation(SELF, ?p, _)) {
        IF (#DID attack(?p) AS ?a) {
            // ask the offending player to cease its attacks
            DROP ?a;
            SAY [to=?p, type=$REQUEST, contents=#I relation(SELF, $ZERO, $ONE)];
        }
        ELSE {
            // improve relations with players which did not attack in
            // the last turn
            CALL adjustRelation(?p, $EIGHTH);
        }
    }

    ADOPT #DID finishInquiries();
}


PROCEDURE handleReply() {
    WHEN [type=$REJECT, from=?X] {
        // the player rejected the peace offer - reduce the relation
        CALL adjustRelation(?X, $NEGQUART);
        RETURN;
    }

    // don't give any credit for accepting the request - the
    // absence of attacks is the only way to improve relations
}

PROCEDURE handleNewMessage() {
    WHEN [type=$REQUEST, from=?P, contents=#I relation(?p, ?l, ?u) AS ?c] {
        // Accept only requests from friendly players
        IF (#B relation(SELF, ?p, ?r), ?r >= $ZERO) {
            IF (?r <= ?l) {
                // always accept to improve relations
                CALL ensureMinimumRelation(?p, ?l);
                SAY [to=?P, type=$ACCEPT, contents=?c];
            }
            ELSE {
                // adjust relations to be as close to the
                // request as possible, but do not go below
                // $ZERO (i.e., don't engage in a new war)
                CALL ensureMaximumRelation(?p, ?u);
                CALL ensureMinimumRelation(?p, $ZERO);
                SAY [to=?P, type=$ACCEPT, contents=?c];
            }
        }
        ELSE {
            SAY [to=?P, type=$REJECT, contents=?c];
        }
     }
}


PROCEDURE giveOrders() {
    XEQ resetFriends();
    IF (#B relation(SELF, ?P, ?R), ?R >= $NEGQUART) {
        XEQ registerFriend(?P);
    }

    IF (!XEQ attack(Q add(0.40, Q random(0.2)))) {
        XEQ defend();
    }
}


PROCEDURE updateStatus() {
    WHEN NOTHING {
        // reset the finished inquiry state
        DROP #DID finishInquiries();
        RETURN;
    }
    WHEN [type=$PLAYER_ELIMINATED, contents=?X] {
        DROP #B relation(?X, _);
        RETURN;
    }
    WHEN [type=$UNIT_ATTACKED, contents=?X] {
        CALL adjustRelation(?X, $NEGHALF);
        ADOPT #DID attack(?X);
        RETURN;
    }
    WHEN [type=$SUPPORT_CENTRE_ATTACKED, contents=?X] {
        CALL adjustRelation(?X, -0.75);
        ADOPT #DID attack(?X);
        RETURN;
    }
    WHEN [type=$SUPPORT_CENTRE_CONQUERED, contents=?X] {
        CALL adjustRelation(?X, -1.5);
        ADOPT #DID attack(?X);
        RETURN;
    }
}
```

C.5 The Cowardly Player

```
/**
 * The cowardly player.
 */
DEFS {
    // the initial relation to all players
    $INITIAL_RELATION = 0.75;
}


PROCEDURE inquire() {
    IF (#B stronger(?P) AS ?x) {
        // try to influence the stronger players to be friendly
        // towards the player
        DROP ?x;
        SAY [to=?P, type=$REQUEST, contents=#I relation(SELF, $ZERO, $ONE)];
    }
}


PROCEDURE handleReply() {
    // the replies are insignificant,
}

PROCEDURE handleNewMessage() {
    WHEN [type=$REQUEST, from=?P, contents=#I relation(?p, ?l, ?u) AS ?c] {
        IF (Q strengthOf(?P) >= Q strengthOf(SELF)) {
            IF (#B relation(SELF, ?p, ?r) AS ?y) {
                // the sender is stronger: try to honour the request,
                // but don't reduce relations to an even stronger
                // player
                IF (Q strengthOf(?p) > Q strengthOf(?P)) {
                    IF (?u > ?r) {
                        DROP ?y;
                        ADOPT #B relation(SELF, ?p, ?u);
                    }
                }
                ELSE {
                    DROP ?y;
                    ADOPT #B relation(SELF, ?p, ?l);
                }
            }
         }

        // always pretend to accept the request
        SAY [to=?P, type=$ACCEPT, contents=?c];
    }
}


PROCEDURE giveOrders() {
    // calculate a new set of friendly players
    XEQ resetFriends();
    ADOPT #I makeNewEnemy();

    IF (#B relation(SELF, ?P, ?R)) {
        IF (?R < $ZERO) {
            // don't make new enemies unless all relations are friendly
            DROP #I makeNewEnemy();
        }
        ELSE {
            XEQ registerFriend(?P);
        }
    }

    IF (#I makeNewEnemy()) {
        // select the weakest player if no other enemies exist
        XEQ unregisterFriend(Q weakestOpponent());
    }


    IF (!XEQ attack(Q add(0.50, Q random(0.15)))) {
        XEQ defend();
    }
}


PROCEDURE updateStatus() {
    WHEN NOTHING {
        // pre-calculate the set of players which are stronger
        // than the current player, and adjust relations accordingly
        DROP #B stronger(_);
        IF (#B relation(SELF, ?P, ?R) AS ?x,
                Q strengthOf(?P) >= Q strengthOf(SELF)) {
            IF (?R < $ZERO) {
                DROP ?x;
                ADOPT #B relation(SELF, ?P, Q add(?R, $QUART));
            }
            ADOPT #B stronger(?P);
        }
        RETURN;
    }
    WHEN [type=$PLAYER_ELIMINATED, contents=?X] {
        DROP #B relation(?X, _);
        RETURN;
    }
    WHEN [type=$UNIT_ATTACKED, contents=?X] {
        CALL adjustRelation(?X, $NEGHALF);
        RETURN;
    }
    WHEN [type=$SUPPORT_CENTRE_ATTACKED, contents=?X] {
        CALL adjustRelation(?X, -0.75);
        RETURN;
    }
    WHEN [type=$SUPPORT_CENTRE_CONQUERED, contents=?X] {
        CALL adjustRelation(?X, $NEGONE);
        RETURN;
    }
}
```

Acknowledgements

Java™ is a trademark of Sun Microsystems, Inc. in the United States and other countries.

Diplomacy® is a registered trademark of Avalon Hill Games, Inc., a Hasbro affiliate.

References

[1] R. S. Cost, T. Finin, Y. Labrou, X. Luan, Y. Peng, I. Soboroff, J. Mayfield, and A. Boughanam. Jackal: A Java-based Tool for Agent Development. In *AAAI-98, Workshop on Tools for Agent Development*, Madison, WI, USA, 1998.

[2] W. Davies and P. Edwards. Agent-K: An integration of AOP and KQML. Technical Report AUCS/TR9406, University of Aberdeen, Department of Computing Science, 1994.
`http://www.csd.abdn.ac.uk/~pedwards/publs/agentk.html.`

[3] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In U. P. Singh, Anando S. Rao, and Michael Wooldridge, editors, *Lecture Notes in Artificial Intelligence*, volume 1365, pages 155–176. Springer-Verlag, 1997. Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages.
`http://users.wmin.ac.uk/~dinverm/papers/dmars.pdf.`

[4] The rules of Diplomacy®, the game of international intrigue. 4th edition, 2000.
`http://www.hasbro.com/instruct/Diplomacy.PDF.`

[5] The Diplomacy Programming Project. See
`http://dept-info.labri.u-bordeaux.fr/~loeb/dpp/index.html.`

[6] GENCON 2000 tournament rules. Tournament rules for Diplomacy.
`http://www.avalonhill.com/tournamentrules.rtf.`

[7] Jaques Ferber. *Multi-Agent Systems.* Addison-Wesley, Reading, MA, USA, 1999.

[8] M. Fisher. A survey of concurrent METATEM – the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic - Proceedings of the First Intemational Conference (LNAI Volume 827)*, pages 480–505, Heidelberg, Germany, 1994. Springer-

Verlag.
http://www.doc.mmu.ac.uk/STAFF/michael/mdf-pubs/ictl94-survey.ps.

[9] Michael Fisher and Michael Wooldridge. On the formal specification and verification of multi-agent systems. *International Journal of Cooperative Information Systems*, 6(1):37–65, 1997.
http://www.elec.qmw.ac.uk/dai/people/mikew/pubs/ijcis.ps.gz.

[10] Foundation for Intelligent Physical Agents. FIPA abstract architecture specification, 2001.
http://www.fipa.org/specs/fipa00001/.

[11] Foundation for Intelligent Physical Agents. FIPA ACL message structure specification, 2001.
http://www.fipa.org/specs/fipa00061/.

[12] Foundation for Intelligent Physical Agents. FIPA communicative act library specification, 2001.
http://www.fipa.org/specs/fipa00037/.

[13] Foundation for Intelligent Physical Agents. FIPA content language library specification, 2001.
http://www.fipa.org/specs/fipa00007/.

[14] Foundation for Intelligent Physical Agents. FIPA interaction protocol library specification, 2001.
http://www.fipa.org/specs/fipa00025/.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object–Oriented Software.* Addison-Wesley, Reading, MA, USA, 1995.

[16] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, July 1994.
http://logic.stanford.edu/papers/agents.dvi.

[17] James Gosling, Bill Joy, Guy L. Steele, Jr., and Gilad Bracha. *The* Java™ *Language Specification, Second Edition.* Addison-Wesley, June 2000.
http://java.sun.com/docs/books/jls/second_edition/.

[18] The DARPA Knowledge Sharing Initiative External Interfaces Expert Group. DRAFT specification of the KQML agent communication lanugage, 1993.
http://www.cs.umbc.edu/kqml/kqmlspec.ps.

[19] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, November 1999.
http://www.cs.uu.nl/people/koenh/autagent.ps.

[20] Koenraad Viktor Hindriks. *Agent Programming Languages: Programming with Mental Models.* PhD thesis, University of Utrecht, February 2001.

`http://www.library.uu.nl/digiarchief/dip/diss/1953134/inhoud.htm.`

[21] JavaCC - The Java Parser Generator.

`http://www.webgain.com/products/java_cc/.`

[22] N. R. Jennings and M. J. Wooldridge. Applications of intelligent agents. In N. R. Jennings and M. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, pages 3–28. Springer-Verlag, 1998.

`http://agents.umbc.edu/introduction/jennings98.pdf.`

[23] Robert Kowalski and Fariba Sadri. An agent architecture that unifies rationality with reactivity. Technical report, Imperial College, Department of Computing, 1997.

[24] Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD, USA, February 1997.

`http://www.cs.umbc.edu/kqml/.`

[25] John McCarthy. Elephant 2000: A programming language based on speech acts. Unpublished draft, 1992.

`http://www-formal.stanford.edu/jmc/elephant.pdf.`

[26] John McCarthy. Ascribing mental qualities to machines. In V. Lifschitz, editor, *Formalization of common sense, papers by John McCarthy.* Ablex, 1990. First published in 1979.

`http://www-formal.stanford.edu/jmc/ascribing.pdf.`

[27] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

`http://www-formal.stanford.edu/jmc/mcchay69.pdf.`

[28] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Modelling Autonomous Agents in a Multi-Agent World*, pages 42–55, 1996.

[29] Jørgen Fischer Nilsson. Data Logic: A Gentle Introduction to Logical Languages, Logical Modeling, Formal Reasoning & Computational Logic for Computer Science & Software Engineering Students. Lecture notes in DTU course 49233/02280: Data Logic, 1999.

[30] Yoav Shoham. AGENT-0: A simple agent language and its interpreter.

In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, volume 2, pages 704–709, Anaheim, CA, USA, July 1991. AAAI Press/MIT Press.

[31] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, March 1993.

[32] S. Rebecca Thomas. *PLACA, an Agent Oriented Programming Language*. Ph.D. dissertation, Stanford University, Computer Science Department, 1993. Also available as technical report CS-STAN-93-1487[1]

[33] S. Rebecca Thomas. The PLACA agent programming language. In Michael J. Wooldridge and Nicholas R. Jennings, editors, *Lecture Notes in Artificial Intelligence*, volume 890, pages 355–370. Springer-Verlag, 1995. Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures and Languages.

[34] Mark C. Torrance and Paul A. Viola. The AGENT-0 manual. Technical Report CS-TR-91-1389, Stanford University, Department of Computer Science, Stanford, CA, USA, April 1991.
`ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/91/1389/CS-TR-91-1389.pdf.`

[35] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
`http://www.csc.liv.ac.uk/~mjw/pubs/ker95.pdf.`

[36] Michael Wooldridge. Intelligent agents. In Gerhard Weiß, editor, *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*, chapter 1. MIT Press, Cambridge, MA, USA, 1999.
`http://www.csc.liv.ac.uk/~mjw/pubs/mas99.pdf.`

[37] Michael Wooldridge and Paolo Ciancarini. Agent-oriented software engineering: The state of the art. In *Agent-Oriented Software Engineering. Lecture Notes in AI*, volume 1957, pages 1–28. Springer-Verlag, Heidelberg, Germany, January 2001.
`http://www.csc.liv.ac.uk/~mjw/pubs/aose2000a.pdf.`

[1] Despite the fact that Rebecca Thomas was contacted three times by email, two times by the author (first, by an informal inquiry in February 2002, which resulted in Thomas offering to send the document; a month later, an inquiry into the reasons for the absence of the document, to which Thomas apologetically repeated her intentions to send an electronic copy) and once by former IMM librarian Petra Dalgaard (who forwarded the original mail with a formal request for the document in June 2002, but did not get any answer) - and despite that the IMM library placed an urgent request for a copy of [32] from the U.S. Library of Congress already in April 2002, none of these efforts have resulted in a copy of the document. Therefore, any assertions about the contents of [32] given in this document are based solely on the secondhand information found in other references on this list, most notably [33].

[38] Michael Wooldridge and Nicholas R. Jennings. Pitfalls of agent-oriented development. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pages 385–391, New York, 1998. ACM Press. http://www.ecs.soton.ac.uk/~nrj/download-files/aa98.ps.

List of Acronyms

ACC	AgentC Compiler
ACL	Agent Communication Language
ACME	AgentC Mental Engine
ACT	AgentC Toolkit
AFC	Agent Foundation Classes
AI	Artificial Intelligence
AOP	Agent-Oriented Programming
API	Application Program Interface
APL	Agent Programming Language
ASCII	American Standard Code for Information Interchange
BDI	Belief, Desire, Intention
BNF	Backus-Naur Form
DAI	Distributed Artificial Intelligence
FIFO	First In, First Out
FIPA	Foundation for Intelligent Physical Agents
GMI	Generic Message Interface
JLS	Java Language Specification
KQML	Knowledge Query and Manipulation Language
MAS	Multi-Agent System
OOP	Object-Oriented Programming
UML	Unified Modelling Language

Index