IT 08 027

Examensarbete 30 hp
Juli 2008

Connecting Soar to HLA

Gunnar Persson

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Connecting Soar to HLA

Gunnar Persson

This Master Thesis report explains the project of trying to connect the human cognition architecture Soar (an AI architecture) to the simulation architecture HLA (High Level Architecture). Descriptions of both Soar and HLA are presented.

A connection between the two, whose main objective is the transfer of relevant data between Soar and HLA, is not a completely straightforward task. The task involves many different aspects of the HLA architecture, including threading in HLA interfaces, dealing with the HLA data representation, time management, transfer of ownership of HLA objects, HLA interactions, etc.

Analysis of the different aspects and their effects on a Soar component has been conducted and is presented. Different possible approaches to solutions to problems and their respective advantages are briefly described.

An implementation of a system including the most vital functions has been created. The system itself, some problems that came up during the implementation of the system and the actions taken to circumvent them, are described together with satisfactory results.

The paper ends with a discussion of different problems that includes some ideas of how to move forward.

The report concludes that creating a system that can deal with most of the aspects (inherent in the HLA architecture) in a general manner is probably too complex. With a more specified behaviour that a user will have to be confined to, a reliable system could be created. The report also concludes that in order to create a reusable HLA-connected system that not only works with Soar but with other AI architectures as well, the design of the system will probably have to be created with reusability plans in mind from the very beginning of the design process.

Handledare: Niklas Wallin
Ämnesgranskare: Bengt Jonsson
Examinator: Anders Jansson
IT 08 027
Tryckt av: Reprocentralen ITC

Chapter 1

Preface

This master's degree thesis was carried out as part of the Computer Science Program at the Division of Computer Systems, Department of Information Technology, Uppsala University. The work was conducted at the Swedish Defense Research Agency (FOI), in Stockholm, with Niklas Wallin as supervisor. The examiner was Professor Bengt Jonsson.

At FOI, this thesis work is included in the project "Computer Generated Forces (CGF)". CGFs can play a part in simulations for the use of training, education and analysis. The purpose of the project is to explore possibilities and conditions to construct a library with models of CGFs representing military and non-military units at different levels of aggregation, to compare the usefulness of different architectures in CGFs, and to develop user friendly interfaces to AI (Artificial Intelligence) architectures being used.

When working with HLA, pRTI$^{\text{TM}}$, version 1.3 from Pitch was used. The Soar interface used was Agent Interface (created by Niklas Wallin at FOI). The implementation part of the thesis work was written in C++ code on a Linux platform.

Contents

Chapter 2

Introduction

2.1 Background

For simulations to be as realistic as possible, they should mirror the real world in a correct manner. In order to achieve this, it is often useful to include human behaviour in the simulations. Modeling and adding human behaviour to simulations constitutes a substantial part of the CGF project at FOI. By using architectures specialized for human cognition, the work of modeling human behaviour is simplified. Soar is one such cognitive architecture, and of all existing cognitive architectures it is the mostly spread.

Constructing and running simulations are also simplified by using an appropriate architecture. High Level Architecture (HLA) is the most established software architecture for simulations today, particularly in the defense area. It is common to have different simulations cooperate, and thus creating a larger simulation system. HLA makes the participating simulations inter-operable when performing distributed simulation, and combines them into a larger system. Another advantage of using HLA is that it encourages and simplifies reuse of participating simulations, making the process of designing and creating simulations more efficient.

Utilizing Soar-connected software in HLA-based applications will enable human behaviour being modeled in larger simulation systems.

2.2 Purpose

Speaking in HLA terms, all participating simulation components that are part of a larger simulation system are called *federates.* Transforming a regular simulation to become a federate is not completely straightforward. The simulation must adapt to the design and the "model vocabulary" of the larger simulation system. This task may or may not be, depending on the nature of the simulation, quite comprehensive. The same applies for when a Soar-based component is to be used as a federate. This means that each time Soar is connected to

HLA, additional adaptation work is required. The amount of work required to connect a Soar-based component to HLA would be substantially reduced if a pre-created connection framework already exists.

The purpose of this thesis is to explore the possibilities of creating such a framework, and if possible, to implement it. The result would be a system that can provide an interface with general methods to the user. These methods will automatically take care of most of the tedious adaptational work in order to transform a Soar-based component to become a HLA-compatible federate. The goal is to be able to construct a library of various functions dealing with the Soar-HLA connection. This library enables creating a Soar plug-and-play component to be used in HLA-based simulations. The library should be scalable and reusable.

Soar is not the only architecture for human cognition. Depending on the nature of the HLA-based simulation at hand, there may be situations where using another architecture than Soar is more desirable. Because most human cognition architectures use the same "sense-decide-act" methodology as Soar in its functionality, it would be useful to have a system connected to the HLA that can be reused and appropriate for these other architectures. The system could then be constructed in different layers, and by replacing a layer, making the connection appropriate for another architecture. To find out whether it is possible or not to create such a reusable system is a task that belongs to the thesis.

2.3 Constraints

The work this thesis is based on is restricted to the Soar-HLA connection only, i.e. how to transfer information between Soar and the HLA. It has not been involved in creating intelligent Soar programs or developing sophisticated simulation systems. (Simple Soar programs and overly simplified "simulation systems" have been created for the use of testing only.) Furthermore, some topics have been omitted. For example, Soar has a Learning mechanism which have not been involved in this thesis, neither has HLA's Data Distribution Management (DDM)[1].

2.4 Methodology

When working with HLA, the main information exchange comes in the form of *objects* and their belonging *attributes*.[2] Soar, on the other hand, uses a *Working Memory* (WM)[3] as data source. Interfaces exist that are able to modify Soar's WM at runtime.

The main idea with a Soar-HLA connection system is that whenever modifications are made to objects in a HLA-based application, corresponding modi-

[1]DDM is briefly described in section 4.6.6 "Data Distribution Management (DDM)"

[2]HLA objects and attributes are more thoroughly described in section 4.5.1 "Objects"

[3]Soar's WM is further described in section 3.2.1 "Working Memory" and section 3.6 "Working Memory Representation in Soar"

fications should take place in Soar's WM. If objects are added in HLA, corresponding data should be added in Soar's WM. If objects are deleted from the HLA, data corresponding to these objects should be deleted in Soar's WM. If attribute values are changed in HLA, corresponding values should be changed in Soar's WM, and so on. Similarly (in the other direction), whenever Soar issues any output values it should result in value changes of object attributes in the HLA-based application. By constructing a system in this way, Soar is immediately aware of changes in the HLA-based simulation, and the simulation will immediately record decisions taken by Soar. By including such general functions that automatically take care of the information transfer between Soar and the HLA at runtime, Soar is integrated with and runs together with the HLA-based simulation. The simulation needs not to be interrupted (perhaps manually) just because of the information transfer. Also, having general functions eliminates the need to write session-specific functions each time the system is run.

A system that connected Soar with HLA was implemented. This implementation task did not come without difficulties, so some analysis work prior to the implementation was demanded. In some cases, different alternative solutions were considered and analysed. Some of the issues dealt with were:

- **Associating HLA objects and attributes with the Soar WM structure.**
  HLA objects and attributes need to be associated with the structure in Soar's WM for Soar to be able to reason about them. A natural predefined association does not exist. It can be constructed either through a predefined protocol or (more desirable) guided by a user's choice. The association can be defined in a variety of ways. Each version of a solution will most likely include a tradeoff between different advantageous/disadvantageous characteristics.

- **Converting between representations of data.**
  HLA has an object representation and Soar has a symbolic representation of data. Conversion of the representation needs to be done every time data is being transferred between Soar and the HLA.

- **HLA attributes representing complex datatypes.**
  A HLA attribute may represent a more complex datatype (with its composite parts revealing simpler datatypes). If a Soar-HLA connection system is confronted with such a HLA attribute, it becomes a matter of how to decompose the attribute in order to fit it into Soar's WM structure.

- **Design issues.**
  Design of a Soar-HLA connection system will depend on several circumstances. For instance, Soar works in a cycle of receiving input and giving out output. The information transfer to and from a HLA application system will (most of the time) also be implemented as a cycle. Design will have to take into account that the two cycles are connected properly in order to function satisfactory. Other issues that affect design are replacement of Soar interfaces and HLA interfaces, the use of threaded interfaces leading to potential simultaneous access of data, how user functions are to be used, and designing for reusability purposes.

- **Ownership transfer of HLA object attributes.**
  Working with HLA may include what is called transfer of ownership of object attributes.[4] Inclusion of this feature will most likely affect a Soar-HLA connection system.

- **Including Time Management.**
  If HLA is used in a distributed environment, it is common to use Time Management.[5] Time Management allows a federate to use a logical clock in order to determine in what order events should be received from other federates. Inclusion of Time Management will affect a Soar-HLA connection system.

- **HLA Interactions.**
  Apart from objects and attributes, another form of data that HLA deals with are *interactions*.[6] An interaction represents an event at a certain point in time. Soar's WM structure is not completely equipped to deal with data with such a meaning, which makes the integration of interactions with Soar more complicated.

When creating the system, the idea was to first create a quite simple system containing the most basic functions, like getting the data transfer between Soar and HLA in working order. If this was successful, more and more functionality would be included in the system. Unfortunately, due to time pressure, it was not possible to include some of the issues mentioned above, therefore the system stayed fairly basic. For example, if time had been more generous, a system including Time Management (no pun intended) would have been implemented. However, the thoughts and ideas of how to move forward with these issues are expressed in the discussion part of the thesis.

2.5 An Example Applied to HLA and to Soar

In order to illuminate the problem presentation of this project, a small example game is hereby shown. The game is described in section 2.5.1. A possible HLA simulation model of this game together with an "execution itinerary" to be able to deal with the game in HLA, is described in section 2.5.2. Section 2.5.3 similarly gives an "execution itinerary" of what needs to be done when applying the game to Soar.

2.5.1 The Rabbit And Fox-Game

The game consists of a 2-dimensional maze-like structure, see figure 2.1. Inside the structure, there is a rabbit and a fox moving around. The rabbit should not be caught by the fox, else it suffers. The object of the game is to control the rabbit's movements in such a way as to always try to avoid the fox.

[4]HLA's transfer of ownership of object attributes is further described in section 4.6.4 "Ownership Management"

[5]HLA's Time Management is further described in section 4.6.5 "Time Management"

[6]Interactions in HLA is further described in section 4.5.6 "Interactions"

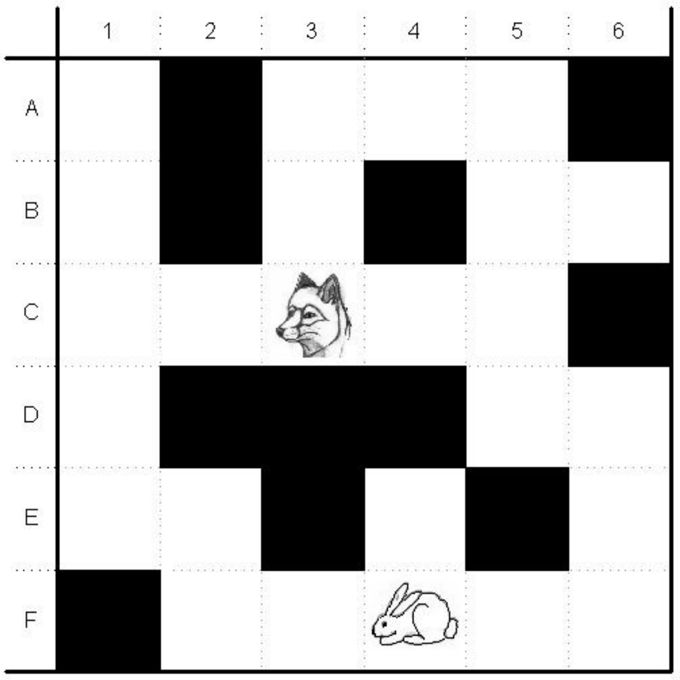

Figure 2.1: Outline of the Rabbit And Fox-game

Information of the status of the game is given out to the user: a) The position of the fox, and b) The position of the rabbit. In figure 2.1, the fox's position is C-3, and the rabbit's position is F-4.

The user can give instructions to the game: A command stating in which direction the rabbit should move. (This command can be either of: a) Move north, b) Move south, c) Move east, d) Move west, or e) Stand still)

2.5.2 The Rabbit And Fox-Game for HLA

This section briefly explains what needs to be done in order to manage a "Rabbit-federate" in the "Rabbit And Fox-federation"[7] in HLA. The main task here is to receive data from HLA (the positions of the rabbit and of the fox) and send data to HLA (the command to the rabbit), see figure 2.2.

HLA simulation model of the Rabbit And Fox-game

HLA objects modeling the state of the game could be designed as follows, depicted in figure 2.3:

The fox is represented as an object of the `Fox` class[8] and has an attribute, `Position`. The rabbit is represented as an object of the `Rabbit` class, and has

[7]The term federation is described in section 4.3 "Federations and Federates"

[8]HLA object classes are described in section 4.5.1 "Objects"

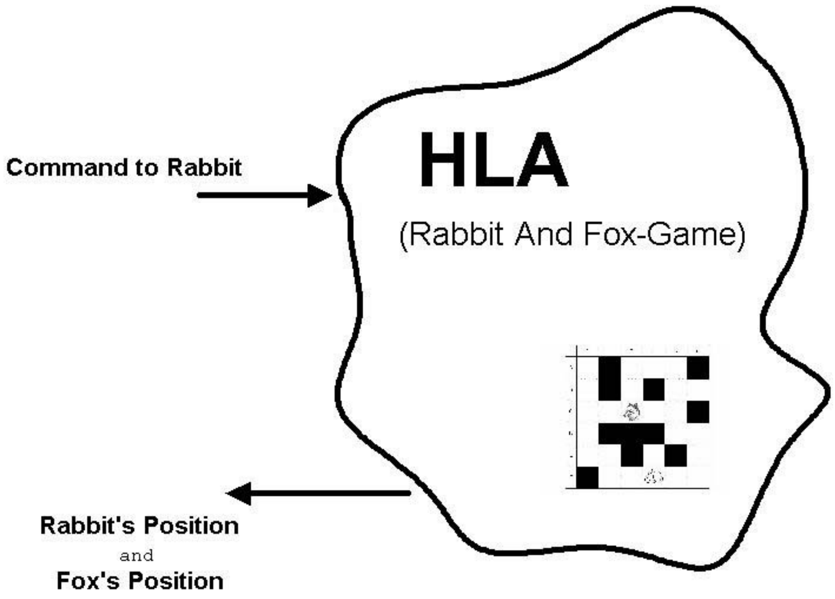

Figure 2.2: The Rabbit And Fox-Game applied to HLA

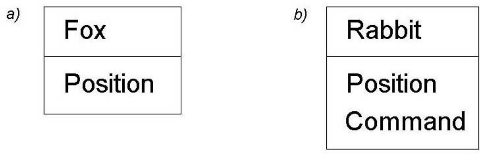

Figure 2.3: HLA classes used in the Rabbit And Fox-game: a) The Fox class b) The Rabbit class

an attribute, `Position`. The `Rabbit` class *also* has an attribute, `Command` (which tells the rabbit the direction it should move in next).

A `Fox` object and a `Rabbit` object will be created as soon as the game starts. A user of the game should subscribe to the attributes `Fox:Position` and `Rabbit:Position`. The user should also publish the attribute `Rabbit:Command`.[9] The user then listens to value updates of `Fox:Position` and `Rabbit:Position`, decides on where the rabbit should move next, and then updates `Rabbit:Command`. By doing this, the federation will make sure the rabbit moves to a new position (or does not move at all), perhaps also make sure that the fox will move to a new position, and thus again update the values of `Fox:Position` and `Rabbit:Position`.

The federation executes in a cycle. The times when we receive the information from the federation in the form of attribute value updates, is restricted to

[9]Publishing and subscribing to HLA attributes is described in detail in section 4.5.3 "Publishing and Subscribing"

certain phases of the cycle. Likewise, the times when we can update attribute values *ourselves* is restricted to other phases of the cycle.

Execution itinerary

This itinerary describes the essential steps that needs to be gone through.

Initial preparations

- Join the "Rabbit And Fox"-federation. (Or perhaps create the federation if *we* should be the creator.)
- Get handle to HLA object class "Fox".
- Get handle to HLA object class "Rabbit".
- Get handle to HLA class attribute "Fox:Position".
- Get handle to HLA class attribute "Rabbit:Position".
- Get handle to HLA class attribute "Rabbit:Command".
- Subscribe to HLA class attribute "Fox:Position".
- Subscribe to HLA class attribute "Rabbit:Position".
- Publish HLA class attribute "Rabbit:Command".

As the execution is up and running

- Discover an HLA object instance of the `Fox` class. (This will happen early on in the execution.)
- Discover an HLA object instance of the `Rabbit` class. (This will happen early on in the execution.)
- Reflect value update of `Fox:Position`. The value may arrive in a special "HLA-format". It may be required to decode the value to a more suitable format in order to elaborate on it.
- Reflect value update of `Rabbit:Position`. The value may arrive in a special "HLA-format". It may be required to decode the value to a more suitable format in order to elaborate on it.
- Update the value of `Rabbit:Command`. The update procedure may require us to encode the command value to a "HLA-format" used by the federation.

The Rabbit And Fox-game is a very simple example of a HLA simulation. The itinerary would grow longer as the application approaches a more realistic use. If the game is extended with, for instance, *more* than one rabbit or *more* than one fox on the board, additional tasks will have to be performed: Discover new object instances *throughout* the game (each time a new Rabbit/Fox is added); Reflect more attribute value updates (since there are more entities in the game); Update more attribute values (if there are several rabbits in the game); Be told that object instances have been removed in the federation (if a rabbit or a fox dies); Perhaps remove object instances ourselves (if we are responsible for doing this); etc.

2.5.3 The Rabbit And Fox-Game for Soar

Regarding the Rabbit And Fox-Game, Soar acts as the "brain" of the rabbit. I.e., by feeding it the positions of the rabbit and the fox, Soar will decide where the rabbit should move next. It then gives out its decision in the form of a command for the rabbit: the direction it should move in (or if it should stand still). When the game has proceeded, new positions of the rabbit and the fox will be given to Soar again. Soar will elaborate on the new status of the game, give out a new command, and so on. See figure 2.4. Soar works in a cycle, so it can only be given input when the "input phase" in the cycle is reached. Likewise, output from Soar can only be received when the "output phase" in the cycle is reached.

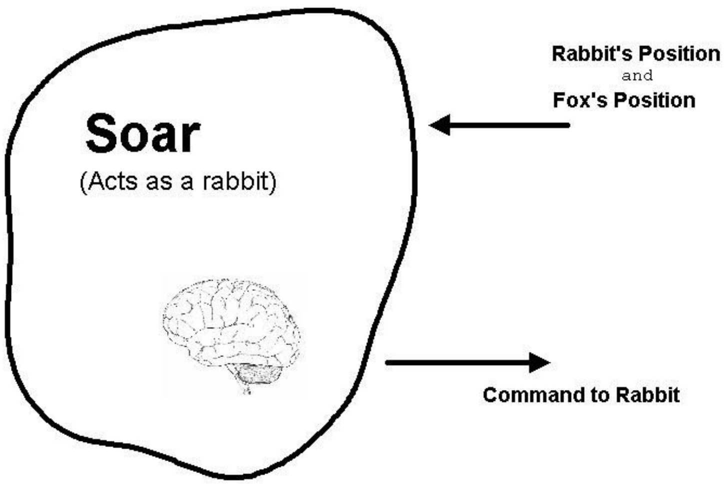

Figure 2.4: The Rabbit And Fox-Game applied to Soar

<u>**Execution itinerary**</u>
This itinerary describes the essential steps that needs to be gone through. Here we assume that, initially, the Soar program does not know of the existence of a Rabbit and a Fox.

Initial preparations

- Create initial input structure in Soar's WM suitable for the Rabbit And Fox-game. Soar will have to have a tailored input structure, in order to properly deal with the value updates that are to come.

As the Soar execution is progressing

- Let Soar know that the Rabbit exists, by adding it to Soar's WM structure. This is done as soon *we* know that the Rabbit exists.
- Let Soar know that the Fox exists, by adding it to Soar's WM structure. This is done as soon *we* know that the Fox exists.
- Give any new position of the Rabbit to Soar. This can only be done when Soar is susceptible to input. We may have to encode the position value to a more suitable format that Soar can understand.
- Give any new position of the Fox to Soar. This can only be done when Soar is susceptible to input. We may have to encode the position value to a more suitable format that Soar can understand.
- Receive output from Soar: I.e. Soar's command to the Rabbit. We may have to decode the given command value to a format that is more suitable for us.

The Rabbit And Fox-game is a very simple example of a Soar application. The itinerary would grow longer as the application approaches a more realistic use. If the game is extended with, for instance, *more* than one rabbit or *more* than one fox on the board, additional tasks will have to be performed: Add more objects to Soar's WM *throughout* the progression of the game (each time a new rabbit or fox appears); Update more position values in Soar's WM (since there are more entities in the game); Receive more command outputs from Soar (if there are several rabbits in the game); Remove objects from Soar's WM (if a rabbit or a fox dies); Perhaps *observe* that Soar has removed objects in its WM (if Soar is responsible for this when a Rabbit dies); etc.

2.6 Outline

Chapter 3 ”Soar” gives a brief description of Soar. Readers already familiar with Soar can skip this chapter.

Chapter 4 ”HLA” describes HLA. Readers already familiar with HLA can skip this chapter.

Chapter 5 ”How the System is Meant to be Used” explains the role for Soar when being used in a HLA-based simulation.

Chapter 6 ”Connecting Soar to HLA, an Analysis” describes in more detail the difficulties with the problem, gives alternatives to solutions, and expresses some opinions on the advantages and disadvantages on them.

Chapter 7 ”Implementation” starts off with briefly explaining the interfaces (to Soar and to HLA) that were used, then describes the implementation of a system. It also highlights some of the problems that came up, and the attempts to circumvent them.

Chapter 8 ”Results” describes the results of the tested implemented system and explains what is left to implement.

Chapter 9 ”Discussion” discusses some of the issues inherent in the problem.

Chapter 10 ”Conclusions” gives some final concluding thoughts to the project as a whole.

Chapter 3

Soar

3.1 Background

Soar[1] is an architecture for human cognition. It has been designed and developed to aid in constructing general intelligent systems. It was created in the early 80's by Allen Newell, John Laird and Paul Rosenbloom, and was first presented in Laird et.al., 1987 [6]. It has been in use since 1983, and has undergone many changes and improvements through the years.

The name Soar was originally an acronym for State Operator And Result, but today it can rather be seen as a name of itself. In the beginning, the aim with Soar was problem solving, but since then more effort has been put on additional topics such as knowledge, learning, planning, robotics, human-computer interaction and cognitive modeling.

The field of human cognition takes its inspiration from many different areas; psychology, philosophy, sociology, cultural and organizational sciences to name a few. This constitutes a problem. Creating a unified theory by sampling what is common from many ”smaller” theories and models from many disciplines is a difficult task. Therefore, more than one unified theory exists. These theories are called Unified Theories of Cognition (UTC)[2]. The creators of Soar do not claim that Soar is a full UTC (and indeed, there are a few cognitive behaviour features that have not been taken into account in the Soar architecture), but it is a candidate UTC. Whether or not Soar can develop to comprise the full set of properties inherent in human cognition at some point the future is not yet known. However, it already involves such important topics as planning, problem solving, goal-oriented behaviour, short- and long-term memory, perception and motor abilities, abilities to select and apply actions to take in the external world or internally in the ”mind”, partitioning higher-level goals into lower-level subgoals, as well as learning.

Some general properties of Soar are the following:

[1]Rosenbloom et.al., 1993 [7], Lehman et.al., 2006 [9], Laird et.al., 2006 [10], Laird, 2006 [11]

[2]Newell, 1990 [12]

- It is a tool for constructing general intelligent systems that work on the full range of tasks expected of an intelligent entity, from highly routine to extremely difficult problems.
- It has a minimized number of distinct architectural mechanisms.
- All decisions are based on relevant knowledge at runtime, instead of being precompiled into uninterruptible sequences.
- It represents and uses appropriate forms of knowledge, such as procedural, declarative, and episodic knowledge, in order to produce appropriate behaviour in the pursuit of goals.
- It has the ability to interact with the outside world.
- It can learn about all aspects of problem solving tasks to perform.

In a study[3] at FOI belonging to the topic of including human behaviour models in CGFs, Soar was examined and compared with one other human cognition architecture named Agent Factory. Soar was considered favourable because i) of its dynamic representations of working and long-term memories, ii) of giving a user a larger freedom to express models in form of objects, iii) of automatic creation of subgoals, iv) of situation of subgoals and operators in working memory, v) and of the possibility to reuse subgoals and operators.

Soar has been used mainly as a problem solving, decision-making tool, to simulate intelligent behaviour in larger systems. By receiving input from the environment, the Soar program — being a part of the environment — can reason about the inputs as well as about the state of its ”internal mind”, elaborate, make a decision, and finally send out one or more outputs to the environment. How to interpret and deal with these outputs is up to the designers of the environment.

For example, a physical robot can be controlled by a Soar program. If the robot can perceive its environment with a camera or some other kind of sensor, and the perceiving data is translated into a format which Soar can handle, the program can deal with this data, elaborate on it and then come to some intelligent decision. The decision might be to move the robot's arm, in which case the output of the program is sent to the robot and the movement is performed.

It is also possible, but less common, to let Soar solve problems independently without interaction with an external environment.

The rest of the sections in this chapter briefly describe the main topics of the Soar architecture, together with references to human behaviour according to human cognition theories. Emphasis has been put on the subjects that directly influence the work of this thesis.

[3]Wallin, 2002 [5]

3.2 Human Memory

Soar contains mechanisms and structure to represent the memory of humans. Humans are dependent on their memory. We use our memory to think, to solve problems, to take actions, and to perceive our outside world. According to the Soar theory of cognition, the human memory falls into two categories: *Working Memory* (WM) and *Long-Term Memory* (LTM). Working memory is sometimes called *Short-Term Memory.* As the names suggest, WM holds information about the current situation, and the LTM holds knowledge that is persistent (unless we forget the knowledge).

3.2.1 Working Memory (WM)

WM represents the current situation for a human.[4] The current situation includes recent events that has happened, and possible events that are about to happen. The current situation is defined *internally* — what is going on in a human mind, and *externally* — what the situation is in the environment the human is currently situated in, i.e. what one can perceive in form of vision, hearing, smelling, etc.

To see the difference between the internal and external representation of the current situation, take as an example that you are solving a mathematical problem. You have made some progress in solving the problem. Suddenly you realize, that to get any further, you need to use a trigonometric formula which is described in your math book. Furthermore, you see with your eyes that the math book is lying in front of you on your desk. Your WM now tells you two things. First, it tells you that to be able to solve the problem, you need to use the trigonometric formula. This fact belongs to the *internal* representation of the current situation. It is something that is going on in your mind, and it is a result of the problem solving progress you have made so far. Secondly, your WM tells you that the math book is situated on your desk in front of you. This is an *external* representation of the current situation. This fact is conveyed to your mind by your perception system of the environment, in this case by your eyes. The way WM in the Soar architecture is represented, there is no difference between either the internal or external description of the current situation.

According to symbolic cognitive theories, the human WM consists of symbols that can be grouped together into objects that represent attributes, events, etc., both internally and in the external environment. The human mind creates and reasons about these symbols, resulting in the human intelligent behaviour. Having a symbolic system is sufficient to simulate intelligent behaviour.

In Soar, WM consists of objects. The objects represent the current situation. The objects can be a concrete representation of something, for example a car. They can also be an abstract representation of something, like the feeling happiness, the status 'finished driving the car', or a state. States are further described in section 3.4 "State and Operator".

[4]Please observe that the following description of WM refers to the Soar theory of cognition. The "real" human WM differs in that it is limited.

The WM in Soar is described in more detail in section 3.6 ”Working Memory Representation in Soar”.

3.2.2 Long-Term Memory (LTM)

In contrast to WM, the LTM in a human mind is where permanent, general knowledge is held. General knowledge is what a human collects and stores through experience and by learning. This knowledge is retrieved whenever necessary to understand the current situation and will directly or indirectly affect any actions taken by the human. For instance, if you are familiar with how to drive a car, you do not need to look up the instruction manual of how to drive a car every time you go for a spin. Instead you use your LTM every time you press the clutch, shift gears, or honk the horn. LTM specifies how to act in certain situations defined by the WM.

WM changes all the time, and depends completely on the current situation. LTM is stable, never changes (except through learning) and exists independently of the current situation. If you are standing in a kitchen cooking a dinner and doing anything *but* driving a car, you still have the knowledge of how to drive a car. That knowledge does not suddenly disappear just because you are doing something else.

The LTM in Soar is described in more detail in section 3.7 ”Long-Term Memory Representation in Soar”.

3.3 Goal-Oriented Behaviour

One mechanism that pervades the Soar architecture is the one that is based on the human goal-oriented behaviour. Humans are always striving to reach a goal. Some goals may require complex and extensive methods — like finishing your university studies, some are smaller and can be achieved in a much shorter time — like cooking a dinner for eating. Even when taking actions that seem trivial, the purpose of taking the action is to achieve a goal. If you duck at a flying object coming toward you, the reason you duck is to avoid injuries (although the action is based on instinct). Whether achieving the goals may be hard and enduring, or trivial and quick, humans basically act in a goal-oriented way all the time — consciously or subconsciously.

Humans also have the ability to divide big goals into smaller goals, or subgoals. The reason for achieving a subgoal is to take you one step further to reaching a bigger goal of which this subgoal is a part. For instance, cooking the dinner may be a subgoal to the bigger goal of throwing a party. Other subgoals belonging to this bigger goal may be to arrange decorations, set the table, fix welcome drinks, and to arrange parking facilities for the guests. Achieving these goals will take you further towards the bigger goal of having a successful party.

In whatever purpose Soar is used, it is always goal-driven, and it includes mechanisms for imitating the human subgoaling behaviour. Structuring goals into subgoals is an important activity for the designer of a Soar program in order to model complex behaviour.

3.4 State and Operator

In order to achieve goals, a human must take some form of actions. By taking an action, the human will (hopefully) get one step closer to reaching the goal. In addition, by taking an action, the human will change its current situation which in turn will affect the choice of the next action to take. At a given point in time, a human is situated in a certain state, and takes an action that modifies the state. The state is defined by features and values determined by the current situation, which in turn is defined by the internal mind of the human and by the external environment. By taking actions in our world which will change any features and values, mentally or physically, we are changing the state we are situated in and move to a new state.

In Soar, this behaviour is implemented as *states* and *operators.* A Soar program is always in a state at a given point in time. An operator applied to a state, modifies the state, i.e. takes the program to a new state.

A state, together with all the features and values that describe the state, reside in WM. Soar always has a top state. Other states are linked to the top state, either directly or indirectly through other states. Also residing in WM are operators, both proposed and selected operators.

According to the Soar theory of cognition, a human does not take random actions. Taking an action is preceded by the process of contemplating what possible options there are, deciding on which option is the most suitable according to the current situation, and *then* taking the action.

In Soar, this is implemented as a three-step process:

1. *Proposal Phase.* Operators that are possible to apply on the state are *proposed* for selection.
   Soar examines the current state by looking at WM, and by referring to LTM which tells Soar how to act in certain situations, Soar will propose a number of operators that are candidates for applying on the state. Referring to human behaviour, this procedure is equivalent to a human coming up with a number of options of actions to take. All possible actions should be considered, even if some may be unsuitable, ill-prepared, or downright stupid. What is important is that it should be *possible* to take the action, and that by taking the action, the human will be taken one step closer to reaching its goal. In Soar, proposed operators are added to the current state in WM.

2. *Selection Phase.* One of the proposed operators is *selected* for application. Soar has a decision procedure to decide which of the proposed operators to select. This procedure is guided by *preferences* among proposed operators. The information about these preferences is defined by the user and resides in a special *preference memory.*

As an example, consider a Soar program controlling a soccer player in a simulated soccer game. The player is in possession of the ball and can either pass the ball to a team-mate or try to score himself by shooting at the opposing goal. If the player sees that the goalkeeper is not prepared for a shot, it could be a good idea to try to score himself rather than to pass the ball. There may be several more proposed operators — like "try to dribble past the opposing defender in front of you", or "take a dive and try to get a free kick" — that Soar needs to take into consideration when selecting an operator. A preference guiding the decision of what to do in this case could be specified as:

"if there is a proposed operator to pass the ball, and
if there is a proposed operator to shoot on goal, and
if the opposing goalkeeper is off balance,
then
the proposed operator to shoot on goal is *better* than
the proposed operator to pass the ball"

The *better* preference is one of the 10 preferences belonging to the Soar language. The preferences provided by Soar are enough to let a user guide the selection procedure in any way he/she may. Two other examples of preferences are the *worst* preference ("do not select this operator, *unless* there are no other proposed operator better than this one") and the *indifferent* preference, enabling random selection of proposed operators.

3. *Application Phase.* The selected operator is *applied* on the current state, taking Soar to a new state.
   When an operator is applied, features and values in WM will change (directly or indirectly), and taking Soar to a new state.

Sometimes Soar comes to a stage when it does not know what to do. In such cases, an *impasse* is reached. An impasse can arise for several reasons:

- No operator has been proposed for selection.
- An operator cannot be selected because specified preferences are not sufficient to pick one operator out.
- An operator cannot be selected because of conflicting preferences.
- An operator has been selected, but that operator has no specified application.

Reaching an impasse is not equivalent to a programming error or system failure. Impasses are a part of Soar's subgoaling and learning mechanisms. When an impasse is reached, Soar automatically creates a *substate*[5]. The substate is added to WM and is linked to the state in which the impasse arose (its superstate). The goal of the substate is to resolve the impasse that caused its creation.

An impasse can also arise in a substate, in which case one more substate is created in WM, linked to the previous substate. There is no limit on the depth of substates. Using substates is a natural way for a designer of a Soar program to create a hierarchy of goals by dividing high-level goals into lower-level subgoals.

[5]State = goal, substate = subgoal

An impasse can be resolved either directly or indirectly. Direct resolving is done if the substate created at the impasse reaches its goal (which is to resolve the impasse) by the normal problem solving activity that Soar performs. Indirect resolving is done if a substate one or more levels below the state in which the impasse arose has modified WM in such a way that the cause of the impasse no longer exists. This can happen, since the problem solving activity in Soar occurs simultaneously at all levels of states. In addition, resolving an impasse can be done if input from Soar's external environment modifies WM in such a way, that the cause of the impasse no longer exists.

In either case, if an impasse is resolved, Soar can continue its reasoning at the exact place where the impasse arose, and therefore no longer needs the substate to make progress. The substate — together with all its substates, and so on — is then automatically removed from WM.

3.5 The Decision Cycle

The human activity consists of constantly perceiving the environment, contemplating on the current situation, deciding on what to do, and taking actions (mentally or physically). It is this activity that is called cognition. In Soar, cognition is implemented as the *Decision Cycle*, and it can be seen as the "motor" of Soar. Figure 3.1 shows the phases of the Decision Cycle.

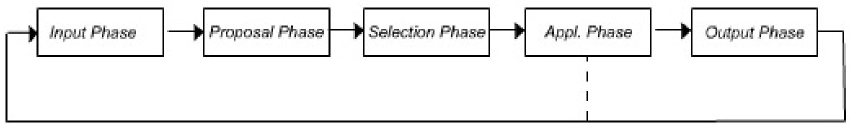

Figure 3.1: The Soar Decision Cycle

Apart from the three phases for applying an operator mentioned in section 3.4 "State and Operator", Soar also has an *Input Phase* and an *Output Phase*. In the Input Phase, Soar receives input (if there is any) from the external environment. This happens just before the Proposal Phase. The Output Phase is responsible for sending out any motor commands that Soar decides to, to the external environment. It happens just after the Application Phase.

Thus, the decision cycle consists of five phases:

1. Input Phase. Receives input from the external world.
2. Proposal Phase. Proposes a number of operators to apply according to the current situation.
3. Selection Phase. Selects one operator to apply.
4. Application Phase. Applies a selected operator.
5. Output Phase. Sends out motor commands to the output system.

If more elaboration is to be done after the Application Phase, Output Phase is skipped in favour of the Input Phase again. Only when no more elaboration is to be done — this is called reaching *quiescence* — Output Phase is entered, and one run of the Decision Cycle is then complete. After the Output Phase, Soar receives input from the external environment again, and the cycle is repeated.

3.6 Working Memory Representation in Soar

As was mentioned in section 3.2.1 "Working Memory", Soar's WM consists of *objects*. An object in WM consists of an *identifier*, a number of *attributes* emanating from the identifier and a number of *values* belonging to each of the attributes. The identifier is the actual object. The attributes usually describe a property of the object or relations with other objects. The value belonging to an attribute is either a *constant* (called *constant value*) or an identifier (called *identifier value*). In other words, an identifier value is another object with its own attributes and values. For a user of a Soar program, a constant value is a symbol with no specific type.

As an example, the object describing a car might have an attribute `color` with the constant value `blue`, an attribute `top-speed` with the constant value `230` and an attribute `belongs-to` with an identifier value being an object describing a person owning the car.

Soar assigns identifiers internal symbols, with names like S1, O2, or N4. The internal representation might be different at different runs of the same Soar program. These symbols are not seen by a user of a Soar program. Instead, identifiers are matched against *variables*. In the Soar language, variables are surrounded by the '<' and '>' sign. Thus, a variable matching the identifier representing the car object above, could descriptively be written as `<car>`.

In the Soar language, an attribute name is preceded by a '^' sign. So, referring to the example above,

```
<car>  ^color       blue
       ^top-speed   230
       ^belongs-to  <person>
```

describes an object with an identifier bound to the variable `<car>`, having an attribute `^color` its value being `blue`, an attribute `^top-speed` its value being `230`, and an attribute `^belongs-to` its value being an identifier bound to the variable `<person>`. An identifier bound to the variable `<person>` will presumably have its own attributes describing the person owning the car.

You can say that an identifier is just a node to keep the structure of the object together. Looking at the identifier alone does not reveal the fact that it is used to describe an object representing a car. That fact is described by the attributes belonging to the identifier. In fact, by knowing only that the above object has a property `^color` with value `blue`, a property `^top-speed` with value `230`, and a property `^belongs-to` with another identifier as value, is not enough to give away that the object is a *car*. Even though the variable `<car>` is used, the properties given above might as well be describing an airplane, a boat, a bicycle, or a very fast smurf. To clarify, the identifier above could have an extra attribute `^type` with the value `car`.

Since attribute values can be an identifier (another object), WM constitutes a hierarchical structure where every identifier is a non-terminal node, the attributes are one-way links, and the constants are terminal nodes with no further links in a non-strict hierarchy. *Non-strict* means that an attribute can be linked to an identifier higher up in the hierarchy. Another word for attribute is *link*.

An identifier-attribute-value triple is called a *Working Memory Element* (WME). Since an object can have more than one attribute emanating from it, the identifier of the object can be part of more than one WME. All WMEs having the same identifier belong to the same object.

As mentioned in section 3.4 "State and Operator", WM includes states. A state is simply an object in WM. The top state is the root node in the WM hierarchy. The state/substate hierarchy is intermingled with the overall WM structure. Additionally, WM contains operators, proposed as well as selected operators.

Figure 3.2 depicts the structure of WM with representation of a car and a person owning the car. S1, N1 and N2 are representations of identifiers as Soar internally might name them. S1 is a state and the top node in the hierarchy, N1 is the identifier representing a car, and N2 is the identifier describing a person. The object with identifier S1 consists of two WMEs, the object with identifier N1 consists of four WMEs, and the object with identifier N2 consists of two WMEs.

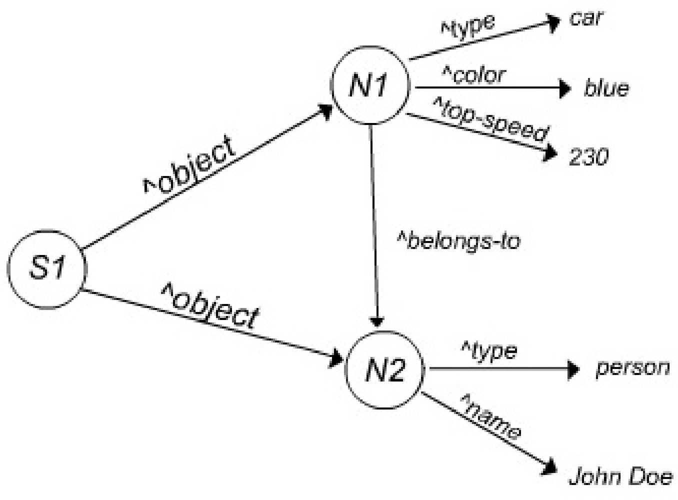

Figure 3.2: An illustration of the WM structure in Soar

WMEs in WM are created in one of three ways:

1. From actions of productions. This is described in section 3.7 "Long-Term Memory Representation in Soar".

2. From Soar's decision procedure that creates states and operators.
3. From the I/O system. This is described in section 3.8 ”I/O in Soar”.

3.7 Long-Term Memory Representation in Soar (LTM)

In order for Soar to be able to solve problems, it must have a representation of Long-Term Memory (LTM). (LTM is also called *Production Memory.*) LTM in Soar is represented as *productions*, or *rules* as they are sometimes called. For the rest of this thesis, the terms *productions* and *rules* will be used interchangeably. The productions can also be seen as the program of Soar, written in the Soar programming language. A Soar program is simply the productions. They are defined by the user and are loaded at the start of a Soar execution. The set of productions do not change during a run of the program. The exception is productions that Soar creates itself when learning. However, the learning mechanism in Soar is beyond the scope of this thesis.

To a beginner Soar programmer, a production would simply seem to be a method of modifying WMEs. But the Soar production language is more sophisticated than that, in that it fits into the Soar theory of cognition concerning states, operators, subgoaling, etc. A Soar production can have one of four functions:

1. Operator Proposal. Proposes a number of operators for selection.
2. Operator Comparison. Guides the selection of operator, by using preferences.
3. Applying Operators. Applies an operator on the state.
4. State Elaborations (monotonic inferences).

The first three functions have already been described. State Elaborations are methods for a programmer to make monotonic inferences (shortcuts) about a state.

The syntax for how to characterize the role of a production to make it fall into one of the four categories above will not be described here, but the interested reader is referred to Laird et.al., 2006 [10] or Laird, 2006 [11].

A production consists of three parts:

1. The name of the production. The name does not affect how a Soar program runs, but exists for descriptive purposes in the same way as the name of a function in traditional programming languages describes what the function does.
2. A set of conditions (also called *Left-Hand-Side*, or LHS).
3. A set of actions (also called *Right-Hand-Side*, or RHS).

The following is an example of a Soar production:

```
sp {add-color*blue
```

```
    (state <s> ^object <obj>)
    (<obj> ^type car)
 -->
    (<obj> ^color blue)}
```

A production is equivalent to an if-then clause in traditional programming languages. If the conditions in the LHS match, the production/rule is *instantiated*, and the actions in the RHS will be performed.

Each condition tests for the existence (or absence) of a single WME or an object in WM. The Soar language provides the user with enough possibilities to express any type of logical combination of WMEs — predicates, negations, conjunctions, disjunctions, etc. As mentioned in section 3.4 "State and Operator" and section 3.6 "Working Memory Representation in Soar", WM consists of states as well as other abstract objects describing the current situation. The first condition in LHS must test for a state. Any further conditions test for WMEs that are directly or indirectly linked to this state.

The conditions in a production uses variable bindings to match against identifiers, attributes, and constants in WM. The two conditions from the production above,

```
   (state <s> ^object <obj>)
   (<obj> ^type car)
```

tests that there is a state in WM having an attribute `^object` with a value (bound to the variable `<obj>`), and that this value is an identifier having an attribute `^type` with the value `car`. All occurrences of a variable in the condition must match WM for the production to match.

If all conditions in LHS match WM, the production is said to *fire*, and the action part (RHS) will be performed. The actions of a production modify WM in almost all cases. The rule in the example above checks for a state having an attribute `^object` with a value being an identifier (`<obj>`) having an attribute `^type` with the value `car`. If there is such a state, and if there is such an identifier belonging to this state, the attribute `^color` with value `blue` is added to this identifier (bound to the variable `<obj>`).

Variables being used for binding identifiers, attributes and constants on the condition side of a production, are bound to the same identifiers, attributes and constants on the action side of the production. So in the example above, the attribute `^color` with value `blue` will be added to any identifier in WM matching the variable `<obj>` on the condition side. If more than one such identifier exists, the attribute will be added to each of these identifiers. Figure 3.3a shows what the structure of WM might be before the above rule has fired, and figure 3.3b shows what the structure might look like after the rule has fired.

3.8 I/O in Soar

Humans have the ability to perceive the surrounding environment and to act in it. In the same way, a Soar program can receive input from the external

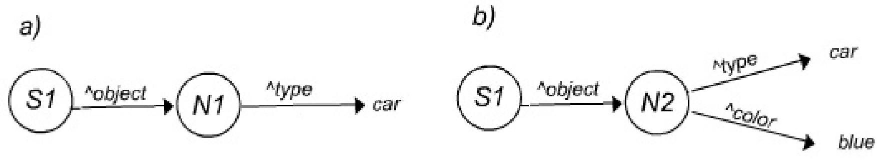

Figure 3.3: a) Part of the WM structure before a rule has fired b) Part of the WM structure after a rule has fired

environment perceived by the perception system, enabling it to reason about the external current situation, and to act in the environment. For Soar, an external environment may be the real world or a simulated world.

WM in Soar can be modified in two ways. One way is through changes to WMEs that are created by application rules, i.e. results from Soar's internal reasoning. Referring to human cognition, these results are equivalent to a human's conclusions after the process of thinking. The other way to modify the WM is by input values from the external environment through Soar's perception system. Soar can control the results from internal reasoning, but not the results from the external environment. The only way to affect the results from the external environment is indirect, if motor commands executed by the program's output system causes changes in the environment, and these changes are later received as input. Other input values may be caused by other sources, like a simulation system, or another Soar program.

A Soar application supplies special *input functions* and *output functions* to simulations in order for them to transfer information to/from Soar. Input functions are called at the start of the decision cycle and transfers data from the outside world to Soar's WM. Output functions are called at the end of the decision cycle and transfers data in the exact opposite direction.

The top state in WM has an attribute `^io`. The `^io` attribute has an identifier which in turn has an attribute `^inputlink`, and an attribute `^outputlink`. The `^inputlink` identifier is where input functions place input in the form of WMEs. The `^outputlink` identifier is where application rules place commands in form of WMEs. Output functions collect the commands and send them to the external environment. Figure 3.4 depicts how `^io`, `^inputlink` and `^outputlink` fit into the WM structure.

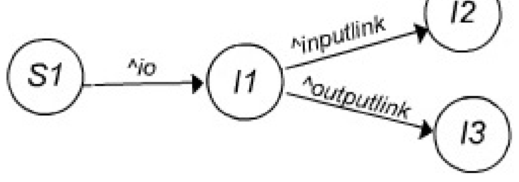

Figure 3.4: I/O in the WM structure

It is common to have an input structure on the `^inputlink` reflecting the external environment. Figure 3.5 shows the simplified input structure of a flight simulation, containing information of an airplane controlled by Soar, and of two enemy planes in the vicinity of the Soar plane.

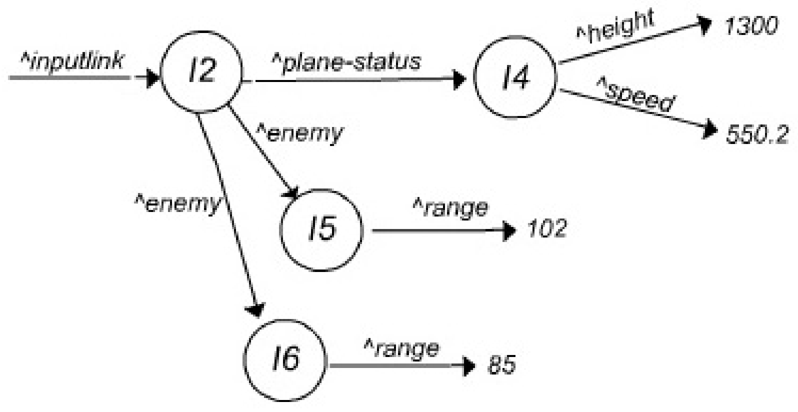

Figure 3.5: Structure of input from a flight simulation

A WME added to the `^inputlink` structure will remain in WM as long as an input function does not remove it. However, a production testing for a single WME on the structure, will only fire once. It will only fire again if the value has been altered, or if the WME is removed and added again (possibly with the same value) by the input function. As an example, consider the `^height` attribute with value `1300` in figure 3.5. As soon as this WME is created, a production testing for this WME will fire once. On subsequent tests of the WME, the production will not fire any more. But if an input function alters the value, the production will fire again. It will also fire again if the WME is removed and added again with the exact same value (`1300`).

WMEs belonging to the `^inputlink` structure should not be added, removed or altered by productions. That should only be done from input functions. Apart from this restriction, a user may deal with the structure the way he/she wants to. Since the `^io` attribute belongs to the top state in WM, it is common practice to use elaboration rules to copy the input structure down to every state in WM. This simplifies writing the conditions in rules, and enables the Soar program to finish dealing with the input before an input function alters the structure again.

WMEs corresponding to output commands are added to the `^outputlink` structure. Output functions collect WMEs belonging to this structure, and send the corresponding command to the external environment.

An output function does not remove a WME corresponding to an output command when the command has been taken care of. The removal should be performed by a Soar production, i.e. it is up to the Soar programmer to make sure that this WME is removed from WM. Otherwise, more and more WMEs corresponding to old commands will be added to WM, wasting space as Soar runs. Removing this WME when taken care of, enables the program to send out the same command again later on, if needed. For this reason, it is common to use elaboration rules that completes this function automatically every time the Output Phase has finished.

Chapter 4

HLA

4.1 Background

Simulations today are used in a wide variety of areas, such as the entertainment industry, as a tool for engineering complex systems, for military purposes, just to name a few. Very often, the desire is to combine simulations into a larger simulation system. For example, simulations representing individual aircraft can be combined with other simulations (airports, radio transmissions, etc.) to create a complete simulation of air traffic control. *HLA* (High Level Architecture)[1] is a software architecture for such simulation systems, including distributed simulation systems.

HLA was created by the simulation community, which demanded a simulation architecture standard. But an essential reason for it to happen was the U.S. Department of Defense (DoD), which sponsored the creation. For military purposes, simulations are used for training, education, and analysis of military operations. By the early 1990's, there were many military simulations around which were not very reusable. Often when a simulation was moved to a new environment, the complete simulation or part of it had to be readapted to fit into the new environment. The need for interoperability and reusability catalysed the creation of HLA. In addition, as HLA developed, industry and IEEE involvement, NATO countries, and academic analysts became stakeholders in the project. HLA is now adopted by the US DoD for use by all its modeling and simulation activities. Furthermore, HLA is increasingly finding civilian appreciation.

4.2 What is HLA?

HLA is a component architecture that "glues" components (simulations) together to form a larger simulation system, where there may be substantial differences in how the different simulations are structured.

[1]Kuhl et.al., 2000 [1], Dahmann et.al., 1998 [2], Dahmann et.al., 1998 [3]

The goals set up when designing HLA were the following:

- It should be possible to decompose a large simulation problem into smaller parts.
- It should be possible to combine smaller simulations into a larger simulation system.
- It should be possible to combine existing smaller simulations with other (maybe unanticipated) simulations to form a new simulation system.
- Functions that are generic to component-based simulation systems should be separated from specific simulations. Resulting generic infrastructure should be reusable from one simulation system to the next.
- Interface between simulations and generic infrastructure should insulate the simulations from changes in the technology used to implement the infrastructure, and insulate the infrastructure from technology in the simulations.

As an architecture, HLA is categorized as being:

- a layered system,
- a data abstraction architecture,
- an event-based, implicit-invocation architecture.

The HLA Standard consists of:

1. *HLA Rules.* 10 rules that expresses design goals and constraints on the simulation system on the whole, on the participating simulations, on the *Object Model Template* (OMT), and on the *Interface Specification.*
2. *Object Model Template* (OMT). The OMT is described more thoroughly in section 4.5.2 "The Federation Object Model".
3. *Interface Specification.* A standardized interface specification between participating simulations and the software that allows the simulation system to execute.

4.3 Federations and Federates

> "The combined simulation system created from the constituent simulations is a *federation.*
> Each simulation that is combined to form a federation is a *federate.*
> A *federation execution* is a session of a federation executing together."
>
> Kuhl et.al., 2000 [1]

The normal role for a federate is to be a regular simulation. A federate may represent simulations at any level of aggregation. It can be a single process or consist of several processes (possibly running on several computers). Traditionally, federates are typically much larger than common software programs, and are rather seen as complete running programs than routines or objects in a library. Quite often, a federate represents a complete simulation system of its own. However, the direction of the current development is that more and more entities in a simple simulation are being used as federates.

A federate can also have other purposes in a federation than those of being a normal simulation. A federate can be a surrogate for live (human) participation in a simulation, a passive collector/logger and displayer of data that is generated as a federation execution progresses, a tool for distributed simulation, a manager of some kind coordinating other participating federates, etc.

A federation contains:

1. A *Runtime Infrastructure* (RTI). The RTI is further described in section 4.4 "The Runtime Infrastructure".
2. A *Federation Object Model* (FOM). The FOM is further described in section 4.5.2 "The Federation Object Model".
3. Some number of federates.

Software elements of a HLA federation are two:

1. Implementation of an RTI
2. Some number of federates.

4.4 The Runtime Infrastructure (RTI)

The federates constituting a federation never communicates directly to each other, no matter how the federation is distributed geographically. Instead, each federate has contact to, and communicates with a Runtime Infrastructure (RTI). The RTI can be seen as a small operative system that works as the medium by which federates exchange information. Figure 4.1 shows a federation containing four federates (two normal simulations, a data logger, and a tool for coordinating the federation) and how they are connected to the RTI. From the perspective of the RTI, a federate is defined as having a single point of attachment to the RTI. The RTI provides the necessary services to a federate for it to be able to successfully participate in the activities of the federation of which it is a member.

The interface between a federate and the RTI is standardized. The RTI can be implemented in a variety of ways as long as it follows the Interface Specification (which belongs to the HLA Standard). The RTI may be connected to a federate participating in the federation or it can be separated from all the federates. It can consist of many processes or as one, and it may require several computers to execute. But conceptually it is seen as one RTI. When designing a federation and participating federates, the only concerns of communication is that to the RTI, and how to deal with the services it provides. There is no need to be concerned with how the RTI works.

Another important aspect is that a federate need not be aware of the existence of other federates. Of course, a federation designer may need to create the correct

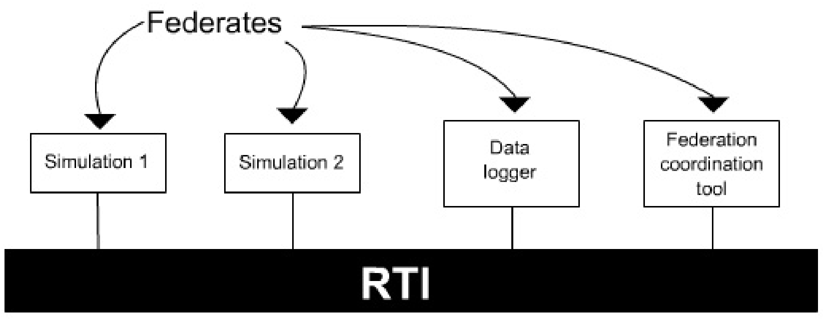

Figure 4.1: Federates connected to the RTI in a federation

number and types of federates, including their relationships with the data used in a federation execution, for the federation to work and run properly. But once a federation execution is up and going, a participating federate need only to deal with the services of the RTI, and with any necessary data the RTI provides.

4.5 Objects and Interactions

The data that federates in a federation use to communicate with each other, comes in forms of *objects* and *interactions.*

4.5.1 Objects

Objects are simulated entities of interest to more than one federate (and thus handled by the RTI) that endure, i.e. persist for some interval of simulated time. The concept of objects (including *object classes* and *object instances*) in HLA has major similarities to that of objects used in object-oriented programming.

Object classes form a strict, single-inheritance hierarchy where each class has exactly one immediate superclass. The root class at the top of the hierarchy — which has no superclass — is named `ObjectRoot`. All classes in the hierarchy are linked to `ObjectRoot`, either directly or indirectly through a series of superclasses.

To each object class belongs a (possibly empty) set of *attributes.* Attributes are inherited from all of a class' superclasses. The *set of available attributes* to an object class are those declared for the class plus the attributes inherited from all of the class' superclasses.

By default, `ObjectRoot` contains one required attribute, `privilegeToDelete-Object`, whose purpose is explained in section 4.5.4 "How Objects are Used". Thus, by inheritance, all objects in the hierarchy contain the class attribute `privilegeToDeleteObject`.

Figure 4.2 shows the hierarchy of object classes. `ObjectRoot` is the top class in the hierarchy. (`RTIprivate` and `Manager` are two classes, with `ObjectRoot` as their direct superclass, that are used for purposes not covered in this report.) The set of available attributes for class `A` are `privilegeToDeleteObject` and `a-attribute`. The set of available attributes for class `B` are `privilegeToDeleteObject`, `a-attribute` and `b-attribute`. The set of available attributes for class `C` are `privilegeToDeleteObject`, `c1-attribute` and `c2-attribute`.

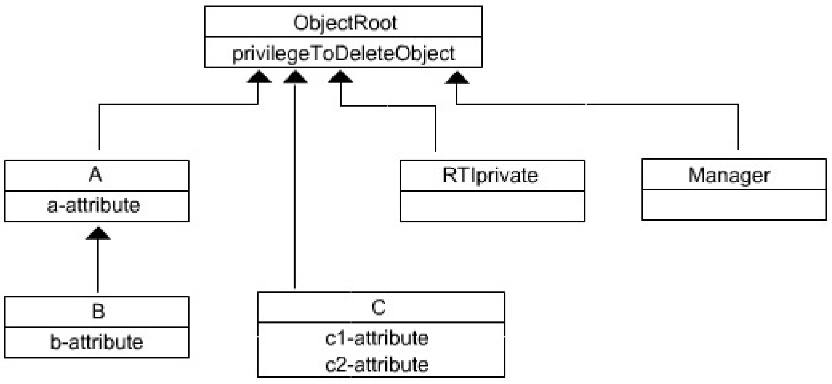

Figure 4.2: HLA object class hierarchy

Running a simulation is almost all about modeling entities. To model an entity means to give it properties and to modify these properties as a simulation progresses. In HLA, this is accomplished by federates registering unique instances of object classes, furnishing values to the attributes of these instances and to update the attribute values as a federation execution is run. Federates interested in attributes belonging to an object are notified every time some federate is issuing an update of these attributes.

Each federate must make some translation from its internal representation of simulated entities to HLA objects as specified in the Federation Object Model, FOM (see the next section). If a federate has been written with the intention of HLA-compliance, the translation may be very straightforward; if a federate is being adapted to the HLA, the translation will be more involved.

4.5.2 The Federation Object Model (FOM)

Information about object and interaction classes used in a federation is situated in what is called a *Federation Object Model* (FOM). It contains the relationship between object classes and their attributes and interaction classes and their *parameters* (parameters are described in section 4.5.6 "Interactions"). The FOM also contains other useful information like, for instance, definitions on transportation type of individual object attributes.

The FOM is created by the federation developer, and forms the vocabulary to be used when federates in a federation exchange data with one another (via the RTI). The FOM must be the same for every federate, and is a parameter to the RTI when a federation execution is started. An FOM must follow the guidelines of an *Object Model Template* (OMT), which is the meta-model for all FOMs. The OMT is part of the HLA standard.

The HLA has been designed to support extension of the FOM, i.e. adding information to the FOM without having to alter the behaviour of federates working with an earlier version of it. One way to extend the FOM is to add object subclasses to the hierarchy of object classes. Another is to add class attributes to already existing object classes. With careful design, a federation designer can add object classes and/or class attributes without affecting the behaviour of federates dealing with an original subset of the classes/attributes.

4.5.3 Publishing and Subscribing

If a federation is large and complex, there can be many object instances present in a federation execution. As a result, a lot of attribute values may be updated. This can cause a heavy traffic between federates and the RTI, which, especially in the case of a distributed federation, reduces bandwidth and processor capacity. To reduce the amount of information being exchanged, the action of publish and subscribe exists. In order to register object instances and to update attribute values, a federate must first declare its interest in doing so by *publishing* the attributes it intends to update. Similarly, in order to be able to discover object instances and to receive updates on attribute values, a federate must first declare its interest in doing so by *subscribing* to the attributes it is interested in. This means that the exchange of data through the RTI can be reduced to what is minimally required.

Another effect of the publish-subscribe function is that federates are protected from changes to the FOM. Even though the FOM is extended with, say, new subclasses to a class whose attributes a federate is subscribing to, the federate need not alter its code in order to deal with new attributes appearing in the federation, attributes the federate originally did not consider in its use of the object class. The RTI makes sure updates on the subscribed attributes are delivered to the federate as if they were attributes of the subscribed class.

Furthermore, the publish-subscribe function:

- decouples producers of data from consumers. Producers (federates publishing attributes) need not consider which federates should receive its attribute updates, and consumers (federates subscribing to attributes) need not consider which federates are updating the attributes they are interested in.
- reduces the complexity of simulations
- allows simulations to be extended easier (extended with extra information in the FOM, or extended with additional federates)
- enables reuse of simulation components

Publications and subscriptions should be made at the beginning of a federation execution. Typically, publications and subscriptions do not change in an execution once they have been made, even though it is possible to do so. The RTI keeps track of which federates are publishing/subscribing to which attributes, and makes sure attribute updates are delivered to the correct recipients.

A federate should publish any class attribute it intends to update. The federate specifies a set of class attributes to publish, together with an object class. The attributes must belong to the set of available attributes of the class. The federate does not have to publish *all* of the attributes declared for a class. It can publish only a subset of the set of available attributes. More than one federate can publish the same class attribute.

If a federate wants to receive updates on a particular attribute, it needs to subscribe to that class attribute. The procedure of subscribing is analogous to that of publishing. It is possible to have a class attribute *both* being published and subscribed to by the same federate.

4.5.4 How Objects are Used

The exchange of data through HLA objects (or — taking another point of view — modeling simulated entities represented as HLA objects), is done in the following way:

When creating an object instance of an object class in the federation, a federate *registers* the instance. To register an object instance, the federate must publish at least one attribute of the set of available attributes belonging to the corresponding object class. It cannot register the instance until it has published any of the attributes. A federate interested in an object, *discovers* the instance by the RTI, when another federate has registered it. To discover an object instance, the federate must subscribe to at least one attribute of the set of available attributes belonging to the corresponding object class. It will not discover the instance until it has subscribed to any of the attributes.

To furnish/refurnish an instance attribute a value, a federate *updates* the attribute. The federate must publish the attribute it updates. Several attributes can be updated at once, considering they belong to the same object instance. A federate can update different attributes at different times, it needs not update all its published attributes at the same time (even if they belong to the same object instance). A federate receiving an instance attribute value (either an initial value or a new value replacing an old) updated by some other federate, *reflects* the attribute by the RTI. To reflect an attribute value, the federate must subscribe to the corresponding class attribute. Several attributes can be reflected at once, if they belong to the same object instance. The federate may not reflect values on *all* attributes it subscribes to. A reflected value replaces any old value the federate might have stored for the attribute.

A federate wishing to end the existence of an object instance in the federation, *deletes* the instance. A federate that has earlier discovered this instance, is then told by the RTI to *remove* the instance.

It can happen that a federate subscribes to attributes of an object class after a corresponding instance has been registered (either because the federate is subscribing late, or has joined the federation late). But the RTI keeps track of all registered instances in the federation, and the federate is assured that it will discover all the appropriate instances retroactively, and will miss none. The discovery will be made directly after the corresponding subscriptions have been made.

A registration of an object instance does not carry any initial values for attributes belonging to the instance. Values for attributes are given the first time a federate updates them. This means that a federate discovering an instance will not receive any initial values for the corresponding attributes, but must wait until any values are reflected.

Registration of instances can occur throughout the federation execution, but typically most instances are registered at the start of the execution when a federate is creating the simulated entities the instances represent.

Attributes, as they are communicated by the RTI, carry no type with them. The RTI handles attribute values as an uninterpreted sequence of bytes: the values are delivered to a reflecting federate as they were supplied by the updating federate. Internally, a federate normally has some type associated with each class attribute: a floating point number, an integer, or even a more complex datatype. It is up to the federation designer to decide on how each federate should encode its internal representation of an attribute to a byte-stream before updating the attribute, and how to decode the byte-stream to its internal representation when reflecting an attribute. The method of use when encoding/decoding attribute values varies, depending on what programming language is used.

As mentioned in section 4.5.1 "Objects", all objects contain one class attribute, `privilegeToDeleteObject`, which is inherited from the class `ObjectRoot`. In order to delete an object instance in the federation, a federate must *own* the instance attribute `privilegeToDeleteObject`. If the federate does not own this instance attribute, it must first acquire ownership of it to be able to delete the instance. See more about owning attributes in section 4.6.4 "Ownership Management".

An object instance is removed from a federation when a federate deletes it explicitly. An instance can also be removed if the responsible federate (a federate owning `privilegeToDeleteObject`) resigns from the federation. In either case, every federate subscribing to at least one of the attributes of the corresponding object class, will be told to remove the instance by the RTI.

4.5.5 Advice About Updating Attribute Values

There is a slight problem with federates updating attribute values. The RTI remembers all object instances registered in a federation execution. However, it does not store any attribute values after they have been delivered to subscribing federates. Suppose one federate registers an object instance, and immediately after updates a belonging attribute value once, and never updates it again throughout the execution. A second federate subscribing to the attribute after this is done (perhaps because it has joined the federation late), will discover the object instance from the RTI, but it will miss the first value update and has no guarantee that it will ever receive a value for the attribute. There are three solutions to this problem:

1. Design federates so that they update attribute values regularly. Using this scheme, the subscribing federate will eventually receive a value for the attribute. However, this method may not always be applicable, and federates will be less reusable if they are moved to a new federation where federates do not behave in this manner.

2. Make sure that all publications and subscriptions that will be issued in the federation execution, are made before any attribute values have been updated. This can be done with phased initialisation, where all participating federates synchronize their actions. See more about this solution in section 4.6.5 "Time Management".

3. Use advice from the RTI to trigger necessary attribute value updates. This method has been used in the implementation part of this thesis. It is therefore described more thoroughly:

   A federate wanting a value for a specific instance attribute, can *request* an attribute value update from the RTI for the specific instance attribute. The RTI remembers object instances and knows which federates are responsible for updating which instance attributes. The responsible federate will then get a call from the RTI, asking it to *provide* an update for the specific attribute value. The responsible federate can then update the attribute in the normal fashion (even though the value included may be the same as the value in a previous update), resulting in the subscribing federate receiving a reflect call with the present attribute value. A "request" can be made for more than one attribute in one call, and a "provide" call can ask for updates on more than one attribute.

   The best way is for a subscribing federate to request updates on instance attributes immediately after an object instance has been discovered, and to which the attributes belong. If a responsible federate follows the "provide"-advice given to it by the RTI and subsequently issues updates, the subscribing federate will quickly receive values (if such values are present) for all attributes it is interested in, and will miss none. Designing federates in this manner enables reusability of federates if they are used in other federations than the one they were originally designed for.

4.5.6 Interactions

In a HLA-based simulation, an *interaction* represents a simulated event that occurs at a point in simulated time. An interaction has no continued existence after it has been used in the federation.

Similar to the case of HLA objects defined in the FOM, are *interaction classes* forming a strict, single-inheritance hierarchy. Instead of attributes, interaction classes carry *parameters*, characterizing the interaction. The parameters are inherited from interaction superclasses. The root of the hierarchy is a class, `InteractionRoot`, which carries no parameter. (Compare with `ObjectRoot` in section 4.5.1 "Objects" which has the default attribute `privilegeToDelete-Object`.)

A federate *sends* an interaction of a particular class. Interested federates then *receives* the interaction.

To send an interaction, the federate must publish the corresponding interaction class. To receive an interaction, the federate must subscribe to the corresponding interaction class. Observe that a federate publishes/subscribes to whole interaction classes, not to individual parameters. (Compare with when dealing with objects where you publish/subscribe to individual object class attributes.)

When a federate sends an interaction of a particular class, it supplies values for some or all of the set of available parameters of the interaction class. The receiving federate will receive the interaction from the RTI, including values for the parameters that were supplied by the sending federate. As with object attributes, parameters carry no type. The RTI handles parameter values as an uninterpreted sequence of bytes.

4.6 Six Management Areas

To ease the use of HLA, the services offered by the RTI (federate-initiated services and RTI-initiated services) are divided into six groups:

1. Federation Management
2. Declaration Management
3. Object Management
4. Ownership Management
5. Time Management
6. Data Distribution Management (DDM)

Services from groups one to three are always required for exchange of data in a federation. Groups four to six can be considered more optional. All groups are independent, services from one group do not interfere with services from another group. The services from all six groups are also designed to ease the use of:

- extending a federation, including extending it with new federates, and extending the FOM,
- reuse of federates.

4.6.1 Federation Management

The main function of *Federation Management* services is to deal with creating and destroying a federation execution, and with joining and resigning from a federation execution. Other functions deal with saving and restoring the state of a federation execution in case it needs to be stored, and to organize federation-wide synchronization points.

4.6.2 Declaration Management

Declaration Management services deal with publications and subscriptions to objects and interactions, i.e. the functions that the RTI uses to control the delivery and reception of information to/from federates. The details of Declaration Management services have already been described in section 4.5 ”Objects and Interactions”.

4.6.3 Object Management

Object Management services deal with the exchange of data in a federation. The services include (among other services): registering/discovering object instances, updating/reflecting attribute values, requesting/providing attribute value updates, deleting/removing object instances, and sending/receiving interactions. The details of Object Management services have already been described in section 4.5 ”Objects and Interactions”.

4.6.4 Ownership Management

Ownership Management services deal with transfer of ownership of instance attributes. An instance attribute comes into existence when the object instance it belongs to is registered by some federate. The federate then owns all instance attributes it is publishing. The other attributes (belonging to the set of available attributes of the corresponding object class) that the federate is not publishing are unowned. An instance attribute is always either owned or unowned. The instance attribute ceases to exist when the instance it belongs to is deleted. A federate that owns an instance attribute is responsible for updating it. Different federates can own different instance attributes belonging to the same object instance. During its existence, the ownership of the attribute can be transferred among federates, but it can only be owned by one federate at a time.

When a federate becomes the owner of an instance attribute, it is said to *acquire* ownership of the attribute. When a federate gives up ownership of an instance attribute, it is said to *divest* ownership of the attribute.

The RTI always ensures that ownership transfers are resolved unambiguously. Transfer of ownership of an instance attribute comes in three forms:

1. Through *abdication*. A federate intentionally divests ownership of the attribute, by either resigning from the federation or by ceasing to publish the attribute. The attribute then becomes unowned.
2. Through *negotiation*. Transfer of ownership is negotiated, which results in ownership of an instance attribute jumping from one federate to another. The negotiation is started either from the divesting federate, called *pushing*, or from the acquiring federate, called *pulling*.

- Pushing.
  The divesting federate informs the RTI which instance attributes it intends to divest ownership of. The RTI knows which other federates are publishing the attributes in question. (Only these publishing federates are eligible for ownership.) It informs those federates which instance attributes are eligible for ownership. These federates can then declare their interest in acquiring ownership.

- Pulling.
  An acquiring federate informs the RTI which instance attributes it intends to acquire ownership of. The RTI knows which federate owns the attributes in question, and asks it to divest ownership of the attributes. The owning federate then either divests or denies divestiture.

   In both cases, if negotiation is successful, the RTI informs the divesting federate it no longer owns the instance attributes, and it informs the acquiring federate it now owns the instance attributes. The transfer of ownership is then completed.

3. Through acquisition *unobtrusively*. A federate asks to acquire ownership of an instance attribute, but only if that attribute is unowned, or the owning federate has declared to divest ownership of it. In this case the acquiring federate acquires ownership immediately.

Attribute ownership comes into light concerning the attribute `privilegeToDeleteObject`, the default attribute belonging to the class `ObjectRoot`. To delete an object instance, a federate must own `privilegeToDeleteObject`, which by default is published by all federates. Ownership of `privilegeToDeleteObject` can be transferred among federates just like any other instance attribute. So, if a federate wish to delete an object instance, and it does not own `privilegeToDeleteObject` for the instance, it must first acquire ownership of it (by any of the ownership transfer forms described above). It can then delete the instance.

In running a simulation system, it is not uncommon to use *shared modeling*, i.e. having an entity being modeled by more than one simulation. It might be a convenient method to evolve the state of the entity. For example, the simulation system of a factory floor might see an industrial component being created by having different "machine" simulations perform different actions on the component. The different simulations modify the state of the component entity cooperatively.

In HLA, there are two ways to perform shared modeling:

1. Two or more federates update different instance attributes that belong to the same object instance.

2. Two or more federates update the same instance attribute, but at different times. The ownership of the attribute circulates during the course of the federation execution. It is this type of shared modeling that the Ownership Management services are designed to handle.

4.6.5 Time Management

Time Management services are designed to make sure that time-stamped events are delivered to federates in causal order, and to enable federates to keep and advance their own logical clock in coordination with other federates. These issues are important for a federation execution to behave reproducibly when executing in a distributed environment.

HLA, being a distributed simulation system, suffers from the same inconveniences that all distributed systems do. Namely, network latencies and varying computational speeds in the participating components. There is no way to predict the exact order in which events are sent and received between the different federates. It can vary from one federation execution to the next. There is quite often a desire to have a federation execution behave *reproducibly*, i.e. given the same starting conditions, two executions of the same federation will produce the same outcome. This cannot be guaranteed if the order of events varies from one execution to the next.

The solution is to make sure that events are ordered in coordination with logical time. In this way, events in a federation execution are sent and received by participating federates in the causally correct order. A second run of the same federation execution will then produce the same order of events, making the execution reproducible.

The goal of the Time Management group of services in HLA are:

1. To allow a federate to advance its logical time in coordination with other federates.
2. To guarantee that events are delivered to a federate in the causally correct order, in particular so that the federate will not receive an event belonging to the federate's "past".

Each federate participating in the time management scheme has its own logical time, which increases as a federation execution proceeds.

In HLA, *sending* an event means to perform any of the following actions:

- updating attribute values,
- sending an interaction,
- deleting an object instance.

These services can carry with them a logical time, which the sending federate includes in the invocation of the service. The event is said to be *time-stamped*, and the sending is of the type *Time-Stamped Order* (TSO). An event with no time-stamp is said to have order type *Receive Order* (RO).

In accordance with the above, *receiving* an event, means:

- reflecting attribute values,
- receiving an interaction,
- removing an object instance.

The RTI guarantees that TSO events are delivered to a receiving federate in increasing time stamp order, and that the federate will never receive a TSO event with a time stamp less than the federate's own logical time.

During a federation execution, a federate involved in the time management scheme continually goes through a cycle of asking the RTI to advance its logical time, and being granted by the RTI to advance its logical time.

Figure 4.3 depicts a state machine for a federate involved in time management. The federate is always in one of the states *Time Granted state* or *Time Advancing state*. In Time Granted state, the federate has a known logical time. At some point, the federate wish to advance its time, and asks the RTI to grant it the advance. The federate then moves to the Time Advancing state. At some point, the RTI grants the advance, and the federate moves back to the Time Granted state. With the grant comes the new logical time the federate may advance to, and the federate can then increase its own logical time to this time. In this way the cycle is repeated, and the federate will see its logical time increase repeatedly as the federation execution progresses.

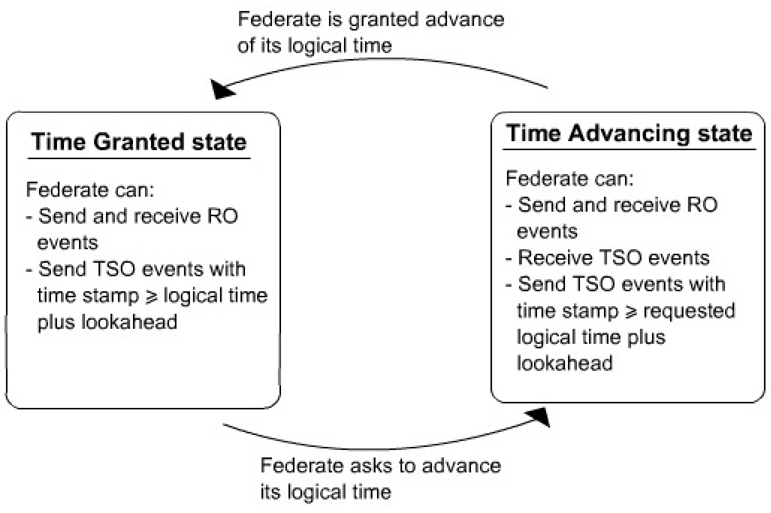
Federate is granted advance of its logical time
Time Granted state
Federate can:
- Send and receive RO events
- Send TSO events with time stamp ⩾ logical time plus lookahead
Time Advancing state
Federate can:
- Send and receive RO events
- Receive TSO events
- Send TSO events with time stamp ⩾ requested logical time plus lookahead
Federate asks to advance its logical time

Figure 4.3: State machine for a time-involved federate

In Time Granted state, the federate can send TSO events with time stamps no less than its own logical time plus a predefined lookahead value[2]. In Time Advancing state, the federate can send TSO events with time stamps no less than its *requested* logical time (the logical time the federate has asked the RTI to advance to) plus lookahead. The federate will receive TSO events in Time Advancing state only. The RTI delivers TSO events in time stamp order, until finally granting the federate the time advance. With the grant, the federate is guaranteed that no event will be delivered to it with a time stamp less than the new logical time it has been granted. RO events can be sent and received in any state.

[2]The reason for using a lookahead value is to avoid deadlock.

The two main methods of advancing time are: 1) *Time-stepped*, and 2) *Event-driven*. The request for advancing logical time works the same for both methods: the federate includes with the request to the RTI the logical time it wish to advance to. The difference between the two methods is seen in how the federate is granted an advance. A time-stepped federate will eventually be allowed to advance its time to the one it asked for. An event-driven federate can be allowed to advance its time to the one it asked for, but it may also be allowed to advance its time to a time *less* than what it asked for. The latter case occurs if the event-driven federate receives an event with a time stamp less than the requested time. The federate is then granted a time equalling the time of this last delivered event. It is up to the federate designer to decide whether to use the time-stepped or the event-driven method. The RTI allows federations with both time-stepped and event-driven federates.

A federate can choose to be *time-constrained* and/or *time-regulating*. To be time-constrained means that the federate's advance of its logical time is constrained by the rest of the federation. To be time-regulating means that the advance of the federate's logical time regulates the rest of the federation. A federate that is both time-constrained and time-regulating is said to be *fully synchronized*. (The description above has assumed fully synchronized federates.) A federate has not yet become time-constrained and/or time-regulating until the appropriate services has been invoked to the RTI.

To have the federation execution being completely reproducible, it is important that participating federates have joined the federation and are taking actions together. These actions include, for example, becoming fully synchronized, publishing and subscribing, and starting to advance time. If a federate joins late, it might miss some events that has already occurred, and the federation execution is no longer reproducible. By using *synchronization points*, which belongs to the Federation Management group of services, it is possible to synchronize the federates so that they take the actions in the different phases together. Using synchronization points is also another way to solve the problem with initial attribute values, i.e. where a late-joining federate might miss attribute values if the updating federate only updates the values once and as soon as it has joined the federation. Instead of using advice from the RTI to retrieve the values, subscribing federates are guaranteed not to miss any initial values if all federates synchronize their actions together.

In order to use synchronization points, one designated federate will have to initiate them. This could be a federate taking part in the simulation, or better, a federate whose sole purpose is to act as a Manager, coordinating the other federates. Having a Manager federate that does not contain federation-specific data is useful, because the Manager can then easily be used in other federations.

A simulation that takes a few hours (or even a few seconds) to execute, can in real time *represent* several days or years. Logical time has no relation to real time, unless such a relation is created. For displaying purposes, it is often useful to *pace* the simulation so that the simulation progress can be observed. The whole simulation, or only the interesting parts of it, can be slowed down if one federate (the obvious candidate is the Manager federate) waits to advance its logical time until the right time in real time has arrived. By doing this, all other federates are impeded, and the federation as a whole executes at a slower pace than if the simulation would be run ”at full capacity”.

4.6.6 Data Distribution Management (DDM)

The publish-and-subscribe functions from the Declaration Management services create certain producer-consumer relationships among federates in a federation. *Data Distribution Management* (DDM) further refines these relationships. By associating with each publication/subscription a *region* from an *abstract routing space*, the RTI is able to restrict the amount of data being delivered to federates to a minimum. DDM is beyond the scope of this thesis.

Chapter 5

How the System is Meant to be Used

As was mentioned earlier, Soar is used as a problem solving, decision-making tool for controlling some component in an external environment, adding human behaviour to the component.

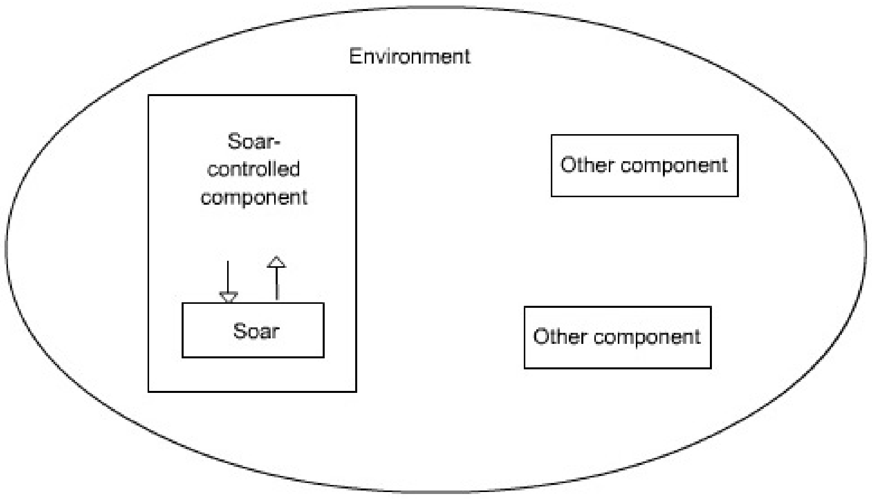

Figure 5.1: Soar hard-connected to the component it is controlling in an environment

The Soar engine could be directly connected to the component, which means that the component would be an intermediary between Soar and the external environment. Figure 5.1 shows an environment with Soar being hard-connected to a component it is controlling, and two other components. However, Soar could also be directly connected to the external environment, detached from the component it is supposed to control. The information exchanged between Soar and the component in order for Soar to control the component would then travel *through* the external environment. Furthermore, there is a possibility to let Soar control several components, all of them being a part of the environment and utilizing the functionality of Soar. Figure 5.2 shows an environment with Soar being detached from two components it is controlling, and one other component.

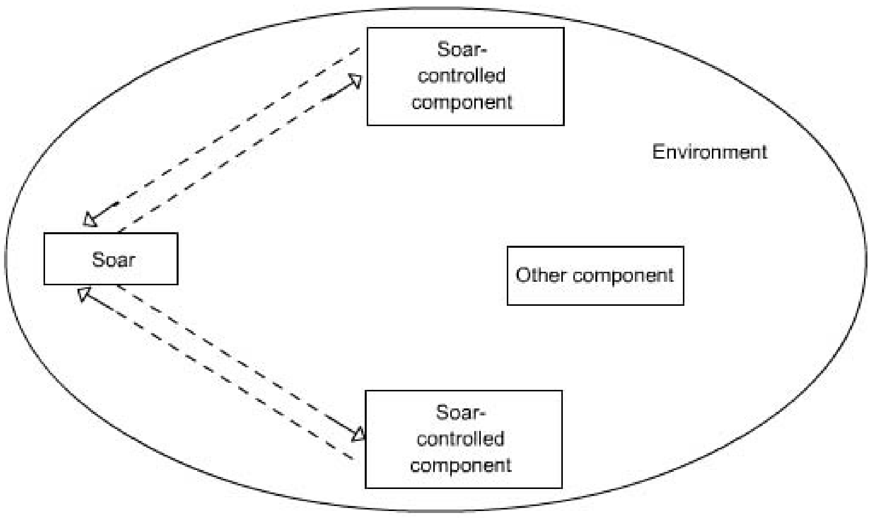

Figure 5.2: Soar detached from two components it is controlling in an environment

Taking the latter approach means that the task for a Soar program would then only be to properly deal with input from the environment and to send out its output to the environment. The behaviour of Soar (the way it acts) is defined by the Soar program (Soar rules). How to deal with Soar's decisions, becomes a task for the components utilizing Soar. Soar's decisions, in form of output, is something that is available in the external environment all the components are operating in.

By not letting Soar be hard-connected to a single component, it becomes more reusable and can be used to control virtually any kind of component, without any additional work connecting it to the component. Integrating Soar in this way with an environment based on component reusability fits well with the ideas on which the HLA was designed.[1]

The work conducted for this thesis has been based on these premises. The only concern for the system is to deal with data from the RTI directed to Soar's WM, and to deal with Soar's output directed to the RTI. How to deal with this output data is completely up to other federates and federation designers. What follows is a short example that, although simplified, hopefully concretises these ideas.

[1]Kuhl et.al., 2000, Ch. 1.2 and 3.2.1 [1]

5.1 A Short Example

Consider two federates that are part of a federation: an aircraft federate, `Aircraft`, and a Soar federate, `SoarAgent`. `Aircraft` publishes and updates attributes of certain HLA objects representing the state of the aircraft (like for instance its position, altitude, and speed). `SoarAgent` subscribes to these attributes and will, every time they are updated (by `Aircraft`), reflect them and direct them to its WM (where there will be an internal representation of the state of the aircraft). Furthermore, the federation contains objects and attributes representing enemy planes in the vicinity of the aircraft. (These objects are controlled by other federates in the federation.) By subscribing to these attributes, `SoarAgent` will also get a picture of the state of the enemy planes in the aircraft's environment. Soar runs its decision cycle and comes to some decision on how `Aircraft` should next react, the decision converted to updates on attributes `SoarAgent` is publishing. (These attributes preferably belong to one or several object instances that `SoarAgent` itself is responsible for and has registered at the start of the federation execution.) By subscribing to these attributes, and thus reflecting them when `SoarAgent` has come to a decision, the `Aircraft` federate will take the actions recommended by Soar. (One of these actions could for example be to move its joystick this way or that, resulting in a new flight course.)

In this way, a relevant behaviour for `AirCraft` is defined by the Soar program. But more importantly, the Soar interface to the RTI has only two main tasks to deal with: 1) Transfer information from Soar to the RTI (by updating instance attributes in the federation) and, 2) Transfer information from the RTI to Soar (by reflecting instance attributes in the federation).

Chapter 6

Connecting Soar to HLA, an Analysis

For the rest of this report, to avoid confusing attributes in Soar's WM with HLA attributes, the term *identifier link* will be used when meaning a Soar identifier attribute, and the term *constant link* will be used when meaning a Soar constant attribute.

6.1 How a User Wants to Deal with the System

A typical federate participating in a typical federation will most likely execute without interruption. Once all required preparations have been made, the federate runs automatically. In this sense, a user's interactions with the system would only deal with preparatory tasks. For example, an essential action is that the user specifies associations between HLA attribute values and positions in Soar's WM where the corresponding values reside. E.g. when an attribute value the system is interested in is reflected, which constant link in Soar's WM should be updated?; and when Soar issues an output command, which HLA attribute value in the federation should be updated as an effect? These registrations of the associations needs to be done before the system starts executing as a federate, and they are most easily done by visible user functions.

When the work this thesis is based on was conducted, there was no specification of a user interface. As can be concluded from chapter 4 "HLA", designing a federate (and also designing a federation, for that matter) and thereby deciding on its behaviour in a federation involves a lot of flexibility, resulting in a wide variety of possible behaviours for the federate. This is one of the advantages of using HLA. For this reason, no extensive analysis has been made regarding a user's interaction with the system.

Some characteristics of the system may however be controlled via user functions in order to configure it to suit the user's preferences. These include variables to run a federation and to run a Soar engine (paths to files to be used, for example).

Additionally, if the system should be involved in Ownership Management and/or Time Management, there will most likely be variables that need to be set before executing (deciding on a time step and lookahead value, for instance).

6.2 Design Issues

No extensive analysis has been made in the case of deciding the ideal way to design a system that connects Soar to HLA, especially when it comes to optimisations of computational capacity and data storage. For reusability purposes however, a few words can be mentioned.

Apart from the connection to a user via user functions, the system is basically connected to two components: 1) A Soar interface, and 2) An interface to the RTI. Following the methodology of modular programming and abstraction, the Soar interface and RTI interface should be encapsulated by modules. This reduces the amount of work of modifying the system if the interface to Soar is replaced with another interface that differs from the original one (and also if the system is to be reused and adapted to some other AI architecture than Soar), and likewise if the interface to the RTI is replaced with another interface. However, replacing the RTI interface will more than likely be a trifle easier than replacing the Soar interface, since any implementation of an RTI follows the Interface Specification which is part of the HLA standard. This results in most interfaces to the RTI being very similar to each other. An interface to Soar does not have the same requirements, resulting in a higher probability that Soar interfaces vary from one another.

The system would require some amount of functionality (including data structures) in order to connect Soar with the RTI. The question is where to place the main functionality of the system. It is perfectly possible to place it in one of the interface modules. Figure 6.1a and 6.1b depict outlines of designs where the main functionality (marked as an encircled capital F) is placed in the Soar interface module and in the RTI interface module, respectively. However, even if the main functionality can reside in an interface module, the more stylistic method would be to separate it from the module completely. One is then assured that replacing an interface would not interfere with the main functionality of the system. This leads to the third approach of placing the main functionality in a separate module of its own, detached from the interface modules. Figure 6.1c depicts this solution. Still, following this last approach, it should be verified whether replacing any of the interfaces will affect the system as a whole in a drastic way.

6.3 Connecting RTI Cycle with Soar Cycle

The Soar engine runs in a cycle of collecting data (Soar Input Phase), elaborating, and giving out data (Soar Output Phase). Any freshly reflected attribute values the Soar program has not been given since the last time it was run should be given to it at the beginning of its cycle. Any output values Soar gives out

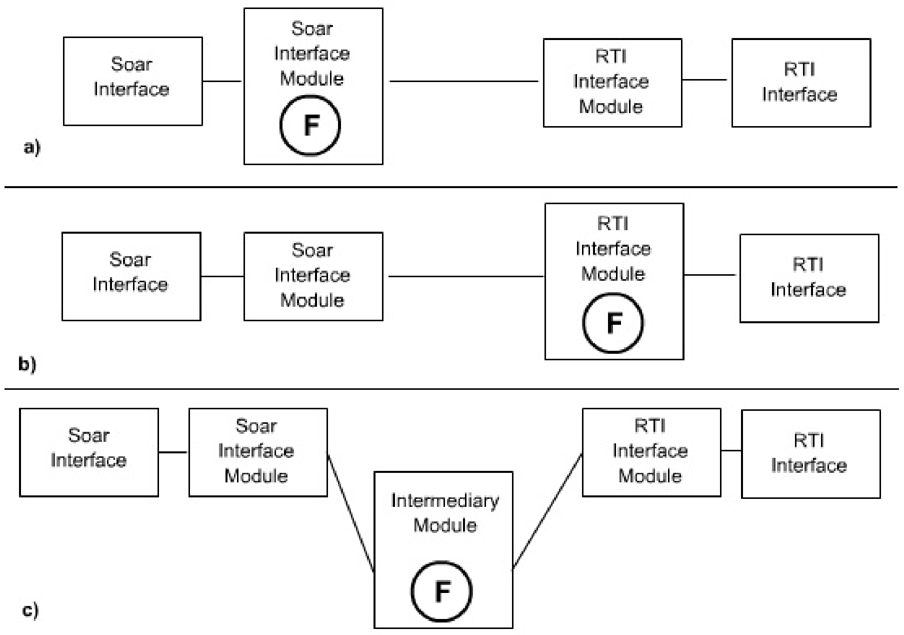

Figure 6.1: a) Main functionality placed in the Soar interface module b) Main functionality placed in the RTI interface module c) Main functionality placed in between the two interface modules

at the end of its cycle should as soon as possible result in updates of relevant attribute values on the RTI.

If the federate is involved in Time management, the code with respect to the RTI also runs in a cycle: moving from Time Granted State to Time Advancing State to Time Granted State, and so on. Connecting this cycle with the Soar cycle is then quite straightforward. The Soar engine should be run while the federate is in Time Granted State. When it has finished, the federate can send out Soar's output values to the RTI, and then ask the RTI to advance its logical time, taking the federate to the Time Advancing State. In Time Advancing State, the federate will receive data from the RTI in time stamp order. A grant of advance of its logical time will take the federate back to Time Granted State again, where the recently received data can be given as input at the beginning of the Soar cycle, which is then run again. This procedure is relevant both for a time-stepped and an event-driven federate.

On the other hand, if the federate is not involved in Time management, data from the RTI may arrive at any moment during program execution. Depending on what interface to the Soar engine is being used, it would basically be possible to insert this data to Soar's WM as soon as it has been received. Soar will continue to elaborate if new input is discovered in its Input Phase. However, if a substantial amount of data is continuously being received from the RTI (this of course highly depends on how the federation is designed and on how other federates behave), a race condition might occur. Soar may not be given a fair chance to reach quiescence and to give out output values, since it is continuously receiving new data in its Input Phase.

A restriction could be made to not insert data to the Soar engine while it is elaborating. Any data received from the RTI during this time can be saved, and then be inserted (and then only) the next time the Soar cycle starts elaborating again. Basically, the federate would then behave in a similar manner as a Time-involved federate, even though it is not.

6.4 Associating HLA Object Representation with Structure in Soar's WM

For Soar to be able to reason about the objects in a federation, these objects need to be associated with the structure in Soar's WM.

6.4.1 Direction: RTI => Soar

Whenever reflecting new attribute values of HLA objects the Soar federate is interested in, these values should in Soar's WM be updated either directly on the `^inputlink`, or on some structure attached to the `^inputlink`.

One issue is how a HLA object instance together with its attributes should be represented on the `^inputlink`. A natural association is that a HLA object instance is a Soar identifier link, and a HLA instance attribute is a Soar constant link (attached to this identifier link). All instances could be put on the `^inputlink`, but a more structured association could also be constructed. Discovered instances may belong to different HLA object classes. Depending on the characteristics of a real-world entity an object class is representing, a number of instances of this class or only one instance of this class may be expected to be discovered. Putting all discovered instances on the `^inputlink` will make it more difficult for a Soar programmer to realize to what HLA object class an identifier link representing a HLA object instance belongs to.

A Soar programmer may want to distinguish between HLA object instances belonging to the same HLA object class. The RTI guarantees that all discovered instances are uniquely defined, and can hence be distinguished from one another. This distinction could easily be translated to an id of each instance, making it possible to add this id as an extra constant link on the identifier link representing the instance. Consequently, the instances can be uniquely defined also inside of a Soar program.

Another issue is to decide *when* an identifier link representing a HLA object instance should be created. An object instance is first discovered. After this, attribute values belonging to this instance will be reflected. Of course, a lot depends on how the other federates in the federation behave, but generally there is no way to predict when an attribute value will be reflected for the first time, if it will *ever* be reflected during a federation execution, or if all or only a subset of the attributes of an instance will be reflected.

An identifier link representing the instance could be created immediately when the instance has been discovered. However, the identifier link will then have no further links (with the exception of a possible ”id-link”) characterizing the instance. A Soar constant link cannot be created without a value, so these links can only be created when a first corresponding attribute value has been reflected from the RTI. The identifier link could also be created when the first attribute value belonging to the instance is reflected, guaranteeing for the Soar programmer that every identifier link representing a HLA object instance will have at least one constant link with a relevant value attached to it. Yet another option is to wait with creating the identifier link until all expected HLA attribute values have been reflected, but since there is no guarantee that this will ever happen during the whole run of a federation execution, this is a more unlikely course of action.

Whenever the identifier link is created, a Soar program can always be constructed to take advantage of the circumstances, and react in a proper way. Some alternatives may result in a necessity to write more Soar rules, other alternatives may result in less Soar rules.

The above has been concluded with the assumption that all HLA attributes to be updated in Soar's WM, can be represented as simple datatypes, like ints, floats, or strings. But a HLA attribute may also represent a more complex datatype (like for instance the composite value of a *position*, consisting of a x-coordinate and a y-coordinate). To associate such an attribute with a number of Soar constant links, and in particular to make it a general option, is a much more complex task. In this thesis, no effort has been made to solve this problem. It has been assumed that the system will only deal with HLA attributes representing simple datatypes.

6.4.2 Direction: Soar => RTI

Whenever an output command has been issued in Soar's Output Phase, the corresponding value should result in an update on a HLA attribute belonging to some HLA object instance on the RTI.

In difference to the case with getting information from RTI to Soar, where an arbitrary number of instances of the same object class could be represented on Soar's `^inputlink`, the system knows beforehand what Soar output value an instance attribute (that should be updated) corresponds to. The natural construct is to let a Soar output value be associated with a HLA attribute belonging to an object instance that the Soar federate has registered itself when starting to execute. The object class that this instance belongs to, could be designated to represent Soar output values only. Possibly, output values could be distributed over several HLA object classes, categorizing for other participating federates in the federation, different *types* of Soar output commands.

Information about the association as such (so called meta-information) needs to be saved somewhere, in order to know where data should be directed during a federation execution. This holds both for transferring data from the RTI to Soar, and vice versa.

6.5 Saving Data

Theoretically, data received from the RTI directed to Soar's WM and vice versa need not to be saved. A reflected attribute value received from the RTI would then immediately result in an update in Soar's WM. An issued Soar output command would immediately result in updates (to the RTI) on attribute values in the federation. However, several facts imply that it is probably a good idea to save HLA attribute values (in both directions):

First of all, not all Soar interfaces would allow insertion to the WM at any time, but only in Soar's Input Phase[1]. Consider a system designed with a functionality that inserts attribute values into Soar's WM immediately after the value has been reflected. If it is later decided that the interface should be replaced with one that does not allow insertions to Soar's WM at any time (or if the system is to be connected to another AI architecture than Soar), then the functionality of the system is no longer adequate for this interface. Saving data will make the system more reusable.

Secondly, designing a system that allows insertion of data into Soar's WM as soon as it is received from the RTI is probably not desirable. If a lot of data is being received and fed to Soar, a race condition might occur where the Soar decision cycle is continuously processing new data in its Input Phase and never (or very seldom) reaches quiescence. A system designed in this way will probably behave in an uncontrolled, un-reproducible manner, and its correct behaviour will be difficult to verify. It would be more desirable to feed the Soar engine with the data that has been received at the moment, to let the decision cycle reach quiescence, and if new data is being received during this time, wait with inserting it until the Soar engine is ready to run again. The only way to do this is to save all new data arriving from the RTI.

Thirdly, information about an association between a HLA attribute and its corresponding position in Soar's WM will have to be used when inserting values in Soar's WM, and when updating attribute values in the federation. So called meta-information for each association needs to be stored somewhere. In this perspective, storing the value itself will not impose a substantial amount of extra work for the system.

Finally, a late-joining or late-subscribing federate may cause the RTI to ask the system to provide updates on attribute values (even if such updates have been issued previously). Trying to retrieve this value directly from the `^outputlink` in Soar's WM is not an option, simply because it cannot be guaranteed that the value is still there. (A Soar programmer is encouraged to remove output commands from the `^outputlink` as soon as the command has been taken care of by an output function.) So the only way to respond nicely to the other

[1]In the implementation part of this thesis a Soar interface, AgentInterface, is used where it is possible to insert values into Soar's WM at any time. It is however not recommended and it is uncertain exactly what happens if this is done.

federate's request to update the attribute value is to save the value. A federation may be designed in such a way that no federates will join or subscribe late, in which case the above circumstances will not occur. But if there is a wish to extend this federation with federates that behave in the above manner, again having a system that saves all issued Soar output commands will make it much more reusable.

6.6 Converting Data from HLA Representation to Valid Soar Types and Vice Versa

Within a Soar program, Soar constant link values are "untyped". But a Soar interface have to associate some kind of type with a constant link value. HLA attribute values reflected from the RTI needs to be *decoded* to a format allowed by the used Soar interface when inserting the values into Soar's WM. Likewise, Soar output commands needs to be *encoded* to the HLA representation of attribute values (a byte-stream) before updating the attributes in the federation.

The type a Soar interface uses for constant link values varies, but most interfaces deal with the three types "int", "float", and "string". How to decode a byte-stream to one of these types, and how to encode one of these types to a byte-stream, depends on how the federation the system is participating in is designed, and on what programming language is used. Transferring the system to new federations may result in new policies on how the encoding/decoding is done. In this sense, it would be an advantage to have functions dealing with the encoding/decoding procedure easily accessible for a user, so that he/she easier can modify the functions to suit his/her needs. If the encoding/decoding part of the system is intricately interweaved with other parts of the system code, it will be more difficult to find and modify it. The best method is probably to have abstract encode/decode-functions easily accessible that a user needs to implement before the system starts executing. In this way the user will have complete control over the encoding/decoding procedure.

Another issue is that of how to save attribute values in the system. They can either be saved in the HLA representation or in the representation the Soar interface uses. For reusability purposes, saving values in a HLA representation would be the better option. If attribute values are saved in the Soar interface representation and if the interface is replaced with an interface dealing with some other representation (or if the system is connected to some other AI architecture than Soar, for that matter), a lot of work will probably have to be carried out in order to adapt the saving of values in the new representation.

6.7 Integrating Time Management into the System

As was mentioned in section 6.3 "Connecting RTI Cycle with Soar Cycle", adapting a Soar federate to involve time management is quite straightforward.

The Soar engine should execute while in Time Granted State, send out its updates of HLA attributes to the RTI, and move to Time Advancing State. In Time Advancing State, the Soar federate reflects attribute values from the RTI, awaits a time advance grant, increases its logical clock, and moves back to Time Granted State again.

A matter to decide is whether the Soar federate should be event-driven or time-stepped. An event-driven federate would reflect attribute values in one invocation with a specified time stamp (or in several invocations if they bear the same time stamp) and then be expected to run the Soar engine. Depending on how the other federates behave, a problem that may arise is that the values reflected bearing one time stamp may be inconsistent for Soar to be able to reason about them properly. Running Soar for a number of cycles in this case would result in no output from Soar or, if Soar rules have not correctly taken this circumstance into account, Soar giving out values that are based on inconsistent input. There is a strong behavioural bond between how a Soar program is written and how the federation and the other federates behave. If only a subset of the expected attribute values have been reflected while Soar is to be run, a more sophisticated system could be constructed that avoids running Soar in this case, hoping that at a next time while the Soar federate is in Time Granted State, all expected attribute values have been reflected, in which case Soar can be run. Using this approach demands some form of protocol to follow that guides the decision of when Soar should be allowed to run. This is something that can be difficult to achieve and requires analysis. Adjusting a Soar program to fit the circumstances (if possible) would most likely seem an easier approach.

A time-stepped federate would reflect attribute values until the logical time the federate has requested from the RTI is reached. This would mean that a Soar federate that is in fact ready to run based on the set of attribute values it has reflected so far may not be able to do so, since it has to wait until the grant of the advance of its logical clock arrives. By the time the advance is granted, the federate may have reflected new attribute values, perhaps ruining the state the federate was in when the running of the Soar engine was more desirable. It is possible to let the time step that a federate uses when it asks the RTI to advance its logical time, vary from one request to another. However, this is hardly helpful since the federate cannot know beforehand when the desired set of attribute values will be reflected. Adjusting a Soar program to suit such circumstances will most likely be more difficult than in the case with the event-driven federate. Nevertheless, hopefully the Soar federate is part of a federation where the other participating federates are fully synchronized and do not run ahead and update attribute values all the time until the Soar federate has responded with an output. This however, may not always be guaranteed. It is a matter of how the federation and the other federates have been designed.

6.8 Ownership Management

The question of whether a Soar federate should be involved in ownership management highly depends on how the federation the Soar federate is participating in is designed. Some circumstances may speak for the inclusion of ownership management. For instance, if the Soar federate shares the modeling of an entity with some other federate, transfer of ownership of instance attributes belonging to the entity has to be done. A transfer of ownership would affect other parts of the system. If the federate is divesting ownership of attributes belonging to some object instance, this fact needs to be conveyed to the position where the information about the instance is kept. Likewise, if the Soar federate is acquiring ownership of attributes belonging to some object instance, it would mean that information about the instance should be modified, possibly created if the system is not previously aware of the instance.

Transfer of ownership follows protocols that are dependent on the behaviour of other federates. To include such protocols in the system, and to be able to offer some kind of general option to a user is a complex task. No extensive analysis to accomplish this has been made in this thesis. An alternative is to allow the user of the system to include code to deal with ownership transfers, i.e. code that responds in a relevant manner if any of the services concerning ownership management issues are invoked on the federate by the RTI.

6.9 Adding Interactions into the System

The main problem with adding interactions into the system seems to be of a semantic rather than technical nature. An interaction can be defined as being a simulated event that occurs at a point in simulated time. Soar does not per se contain mechanisms to represent such an occurrence. Instead, it is the Soar program that, by creating behaviour, must define its own meaning for what an interaction really is. Different Soar programs may interpret the meaning differently. For example, the main question seems to be: For how long should an interaction stay in Soar's WM? Another way of putting it is: When has the "simulated point in time" in which an interaction occurs elapsed? A subsequent question may be: If the "simulated point in time" has elapsed, should the interaction then correctly be removed from WM even though the relevant Soar rules have not yet had the opportunity to elaborate on it? If the Soar program deals with an interaction "late", the meaning of the interaction in the rest of the federation may no longer be relevant.

These concerns are mainly connected with receiving an interaction from the RTI directed to Soar's WM. In the other direction — Soar sending an interaction to the RTI — the procedure seems more straightforward. Soar issues output commands that represents sending an interaction. As soon as possible after this, i.e. the next time calls to the RTI are made, the interaction should be sent.

Once the semantic issues above have been settled, the technical work of adding interactions into the system should not be a difficult task. The analysis and implementation of adding interactions into the system has not been a priority in this thesis.

Chapter 7

Implementation

The work this thesis is based on included the (experimental) implementation of a system called SoHlCo (**So**ar-**HL**A-**Co**nnection). The system deals with HLA objects and their attributes but not with interactions. All data structures used in SoHlCo are implemented as vectors. In this chapter, however, the term *table* is occasionally used to mean a data structure. The system did not include Time management when started, and unfortunately, due to time pressure, it was never integrated. However, from the beginning SoHlCo was designed in such a way that integrating Time management should not impose a substantial amount of extra work.

7.1 Interfaces Used

7.1.1 Agent Interface

Agent Interface is an interface to Soar that was created by Niklas Wallin at FOI. The system created for this thesis chose Agent Interface mainly for its availability. Only an extract of the full functionality of Agent Interface is given here.

Initially, to start up the interface, an object of the `SoarEngine` class is created. This provides a connection to the Soar kernel and enables the user to run one or several Soar agents.

The recommended way to represent a Soar agent is to create an object derived from the `SoarAgent` class. `SoarAgent` contains a member function to load the Soar productions (the Soar program) from a file, and a member function to run the agent for a number of Soar cycles.

Additionally, `SoarAgent` contains three virtual functions that should be implemented in the derived class: `input`, `output`, and `print`. These functions are callback functions invoked by the Soar kernel. `input` is called every time a Soar program reaches its Input Phase. `output` is called every time a Soar program reaches its Output Phase. In `input`, any modifications to structure belonging to Soar's `^inputlink` should be implemented. In `output`, what to do with the data sent out by Soar's motor commands should be implemented. The `print` function is called every time the Soar kernel has any output message to print.

In `input` is where any initial structure on Soar's `^inputlink` should be created. This structure needs only to be created once, and endures during the lifetime of the Soar agent. For this reason, `input` takes as argument a number. This number shows if it is the first time the function is invoked or not. The first time `input` is invoked, initial structure is created on the `^inputlink`. In the subsequent invocations of `input`, any wanted modifications to the structure on the `^inputlink` is made.

An identifier value (or actually an identifier attribute together with its identifier value) is represented by an `IdElement` object. In order to add additional attributes to the identifier, `IdElement` has four different (overloaded) member functions, `IdElement::insert`, that creates (and returns) either:

1) A new `IdElement` object.
2) A `StringElement`, representing a constant attribute (together with the value of type string).
3) A `FloatElement`, representing a constant attribute (together with the value of type float).
4) An `IntElement`, representing a constant attribute (together with the value of type int).

To create a new `IdElement`, the name of the identifier attribute is given. To create a constant attribute, the name of the attribute together with the value of the attribute (a string, a float or an int) is given. It is these given names that will be used within a Soar production to access the identifier and constant attributes.

In this way, a complete structure can be constructed on the `^inputlink` of Soar's WM. The `IdElement` object representing the `^inputlink` can be retrieved by a member function of the `SoarAgent` class: `SoarAgent::getInputLink`.

To change the value of a constant attribute, `StringElement`, `FloatElement` and `IntElement` each has a member function `update` that takes the new value as argument.

It is possible to remove part of the structure created on the `^inputlink`. To remove an identifier or constant attribute, the corresponding object (`IdElement`, `StringElement`, `FloatElement`, or `IntElement` object) is simply deleted. If an identifier attribute is deleted, any further identifier or constant attributes attached to the deleted identifier attribute are also (recursively) deleted.

`output` behaves differently. It takes as argument a pointer (`void *`) to a structure representing Soar's decisions on data sent out by output commands. In order not to have to parse the structure (which is tedious), an `OutputManager` may be used that takes care of the parsing business. This enables a user to associate an output command with a function taking an output value (string, float, or int) as argument. By executing the `OutputManager` in every invocation of `output`, every time an output command has been issued by Soar, an associated function is called, and it is in this function what to do with the output command should be implemented.

For this to work, registrations of these associations needs to be made by the `OutputManager` object. Preferably, these registrations are made the first time the agent's `input` function is called. In the action of registering an association is given the name of the Soar command. It is this name that will be used within a Soar production to add an output command on Soar's `^outputlink`.

7.1.2 RTI Interface

Federate-initiated services are calls to the RTI made from the federate. These services are implemented as methods in a `RTIambassador` object.

RTI-initiated services are calls to a federate made from the RTI. These calls are implemented as pure virtual callback functions in a `FederateAmbassador` object. These methods all need to be implemented. In the class `BaseFederate-Ambassador` — derived from `FederateAmbassador` — the methods have been implemented but with empty function bodies, i.e. they do nothing. To take advantage of RTI-initiated services, a class derived from `BaseFederateAmbassador` should implement (and override) only the methods that are of interest to the federate. This means that calls from the RTI that are not of interest are simply ignored.

Callbacks from the RTI are threaded. Whenever a call to a method of the `FederateAmbassador` is called it will interrupt any other activity being conducted by the federate. The `RTIambassador` object contains two methods: `disableCallbacks()` and `enableCallbacks()`. `disableCallbacks()` blocks all callbacks, forcing the RTI to save them until the call `enableCallbacks()` is called, at which time the RTI will invoke all saved callbacks. This enables the federate to control at what times callbacks from the RTI should be received. At the time of writing, however, these two methods have not yet been implemented. Consequently, since the callbacks run in parallel with the rest of the federate code, care needs to be taken if the callback implementations access data simultaneously as the interrupted federate code.

In order to distinguish HLA object classes etc. from each other, *handles* are used. By making calls to the RTI (via the `RTIambassador` object), a federate may immediately retrieve handles to: an object class, a class attribute, an interaction class, and an interaction parameter. It is wise to make these calls in the beginning of a federation execution, retrieving handles to all the data the federate knows it will use. A handle to an object instance is given when registering or discovering the instance. The handles are unique, and will not change during a federation execution. The handles will constantly be used by a federate when publishing, subscribing, registering an object instance, updating attribute values, reflecting attribute values, etc.

In addition, the interface contains classes and methods to create and use *sets of attributes*, *sets of pairs of attributes and values*, etc. Consequently, calls referring to a set of attributes (for example when updating attribute values) can be made in one invocation instead of in several.

7.2 How the System Acts as a Federate

From a federation point of view, as a federate the SoHlCo system acts in the following way:

Initial preparations
A federation is created (if not already created) and joined to. Handles to object classes and attributes that will both be published and subscribed to, are retrieved. Publications and subscriptions to class attributes are made. Finally, SoHlCo registers instances of classes containing attributes it intends to update later on.

Listing of services used (all federate-initiated):
- Create Federation Execution
- Join Federation Execution
- Get Object Class Handle
- Get Attribute Handle
- Publish Object Class Attributes
- Subscribe Object Class Attributes
- Register Object Instance

When federation execution is running
SoHlCo discovers object instances of object classes that are of interest. Attribute values of these instances are reflected. (These updates will result in updates in Soar's WM.) Since SoHlCo is not a time-involved federate, it requests attribute value updates of newly discovered instances. SoHlCo removes object instances (previously discovered) that has been deleted in the federation.

SoHlCo updates attribute values (that originates from Soar's output commands). It is asked (from other federates) to provide attribute values, even though these values may already have been updated. (These updates are performed the next time calls to the RTI are made.)

Listing of services used:

- Discover Object Instance	(RTI-initiated)
- Reflect Attribute Values	(RTI-initiated)
- Request Attribute Value Update	(federate-initiated)
- Remove Object Instance	(RTI-initiated)
- Update Attribute Values	(federate-initiated)
- Provide Attribute Value Update	(RTI-initiated)

7.3 Overall Design of the System

Figure 7.1 shows the overall structure of the SoHlCo system. The main control of the system resides in an object of the `SoHlCo` class. This class is connected (via pointers) to an object of the `HLAAgent` class and an object of the `HLAFederate` class.

The `HLAAgent` class inherits from `SoarAgent` (from Agent Interface), and works as the interface to Soar. `HLAAgent` implements the functions `input`, `output` and `print`. (`print` has so far only been implemented for debugging reasons.)

`HLAFederate` is the interface module connected to the RTI. It is connected to an object of the `RTIambassador` class to be used for federate-initiated services invoked on the RTI. `HLAFederate` is also directly derived from the class `BaseFederateAmbassador`, and contains overridden functions of the RTI-initiated services that are of interest.

The main functionality of the system as well as all data structures containing the saved data that is being exchanged between Soar and the RTI reside in the `SoHlCo` class. Additionally, SoHlCo contains specified interface functions that a user of the system will want to use.

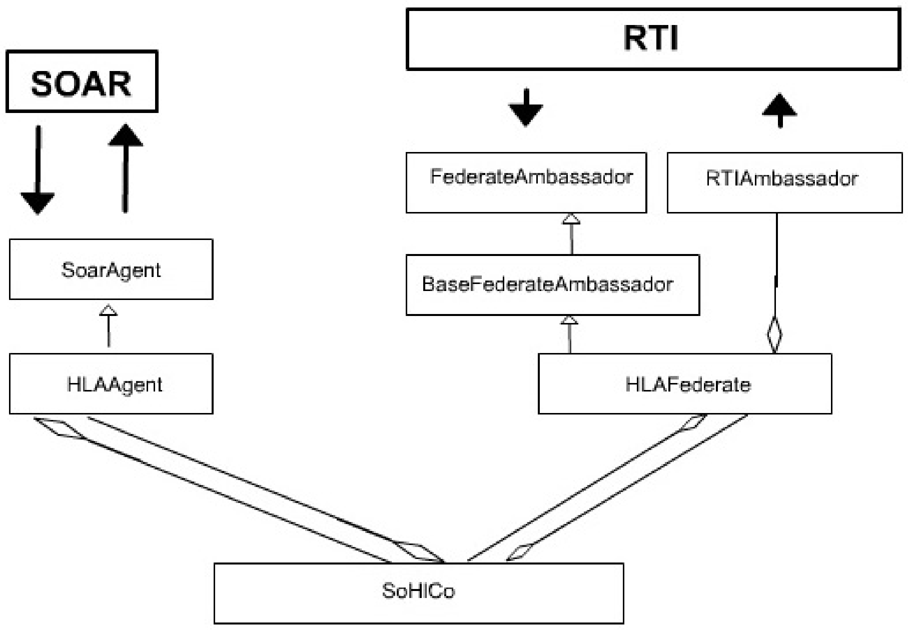

Figure 7.1: The overall structure of SoHlCo

This structure was chosen mainly to separate functionality directly connected to Soar from functionality directly connected to the RTI. In this way, the structure provided an easier look-over of the system, and seemed natural in case more functionality was to be added to the system.

7.4 Connecting Soar Cycle with RTI Cycle

It was decided that even though the SoHlCo federate is not a time-involved federate, it should act in the same cycle (time-advance-grant-cycle) with respect to the RTI in the same fashion as a time-involved federate does. This would make life easier when/if time management was to be included in future work on the system. Consequently, SoHlCo works in a cycle of receiving data from the RTI, feeding Soar with data, letting Soar elaborate, receiving output from Soar, sending out data to the RTI, and then receiving data from the RTI again.

When creating the system, the functionality was divided into the following four phases:

1) "RTI Input Phase"	Receiving data from the RTI
2) "Soar Input Phase"	Sending data to Soar
3) "Soar Output Phase"	Receiving data from Soar
4) "RTI Output Phase"	Sending data to the RTI

Since implementations of `enableCallbacks()` and `disableCallbacks()` in the RTI interface did not exist when SoHlCo was created, data from the RTI may arrive in the form of callback functions at any time during execution. Each time a callback function arrived, all information included in the callback was saved and stored in a callback-queue. This allowed SoHlCo to deal with the callback later on when its RTI Input Phase was reached. In fact, two callback-queues were used. The reason for this was that when reading from and modifying a callback-queue, new callbacks may arrive to the queue. To avoid inconsistencies of the callback-queue resulting from new callbacks arriving to the queue while dealing with it, each time the RTI Input Phase was reached, SoHlCo switched a "callback-queue-flag", allowing it to deal with `callback-queue1` while new callbacks from this time on were added to `callback-queue2`, and vice versa.

The reason why callbacks were saved to be dealt with later in its proper place, is that the callbacks use and modify data structures which are used by other parts of the system. To avoid simultaneous access of these data structures, the resulting implementation of the callback function cannot be allowed to execute until SoHlCo has reached its RTI Input Phase, at which time it is guaranteed that no other part of the system is executing.

7.5 Associating HLA Object Representation with Structure in Soar's WM

7.5.1 Associating HLA Object Representation with Structure in Soar's WM: RTI => Soar

The association was made in the following way:
Each HLA object class the federate is interested in gets an identifier link (hereby called *class-link*) situated on the `^inputlink` in Soar's WM. Every HLA object instance discovered gets its own identifier link (hereby called *instance-link*), which is added to the class-link representing the HLA object class the instance

belongs to. All instances of the same HLA object class (in case several instances of the same HLA object class are discovered) are situated on the same class-link. For each HLA attribute, there is a constant link with value, situated on the instance-link corresponding to the object instance the attribute belongs to. An extra constant link (of type int, hereby called *id-link*) added to the instance-link serves as an id of the instance.

An example will illustrate the association: The system subscribes to attributes `speed` and `height` of the object class `EnemyPlane`. Figure 7.2 shows what the structure in Soar's WM looks like after two instances of the `EnemyPlane` class have been discovered and attribute values for `speed` and `height` for both instances have been reflected. For simplicity, only the relevant part of the structure is shown and Soar's internal symbols for identifiers have been omitted. The class-link for the `EnemyPlane` class has been named `^enemy_plane_class` and the instance-link for an instance of the EnemyPlane class has been named `^enemy_plane_instance`. The constant links for the attributes `speed` and `height` have conveniently been named `^speed` and `^height`.

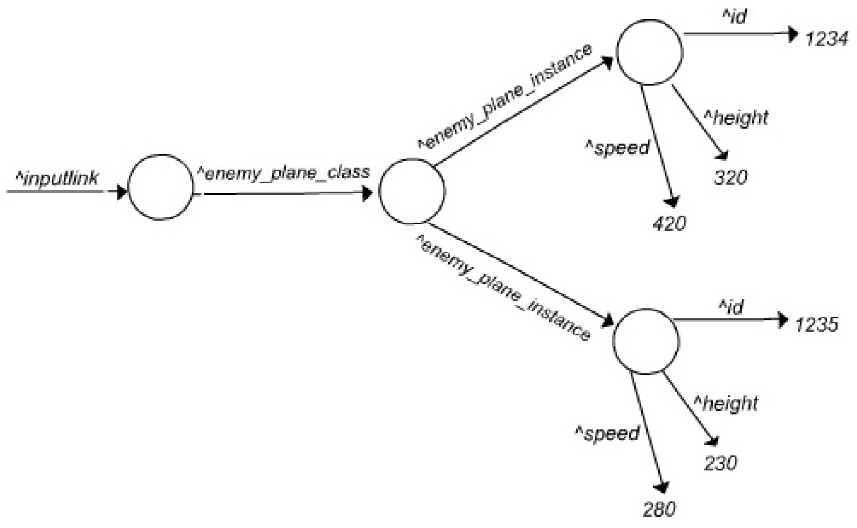

Figure 7.2: The simplified structure of Soar's WM after two HLA object instances have been discovered and attribute values reflected

The class-links are constructed when the initial structure on WM's `^inputlink` is created. An instance-link is created as soon as a corresponding HLA object instance has been discovered. At the same time the corresponding id-link is created. The constant links representing HLA attribute values are added to the instance-link as soon as they have been reflected from the RTI for the first time.

This means that a Soar program, that in its Soar rules expects a number of constant links (with HLA attribute values) situated on an instance-link, may see that not all of those links have yet been created at a certain point in time. This fact is something that has to be considered when writing the Soar rules. There is no way to tell when (if ever) the full set of expected constant links are created. This completely depends on the behaviour of the federates responsible for updating the corresponding HLA attributes.

7.5.2 Associating HLA Object Representation with Structure in Soar's WM: Soar => RTI

Each type of Soar output command is associated with a HLA attribute belonging to a HLA object instance that has been registered by SoHlCo in the initial phase of the federation execution. An issued output command results later on in the RTI Output Phase in an update of the corresponding instance attribute. SoHlCo needs not concern itself with whether it is the first time the HLA attribute value is being updated or not — this is the concern of the other federates interested in the attribute. In either case, the attribute is simply updated.

7.6 User Functions

SoHlCo Constructor
First of all, the `SoHlCo` constructor takes 8 parameters to define various characteristics of the system as a federate:
- Name of file with Soar productions
- Name of federation execution
- Name of FED(Federation Execution Data)-file which contains the FOM of the federation execution
- Name of the Soar agent
- Maximum number of Soar cycles the Soar engine should run if no output is made
- Name of the host of the federation
- Port number the federation runs on
- Federate type (a name that for this implementation has no significant effect)

---

5 user functions were implemented:
`insert_soar_link/2`
`register_soar_input/6`
`register_soar_input_instance_unique/6`
`register_soar_output/4`
`run_execution/0`

insert_soar_link/2
One user function, `insert_soar_link/2` is used to insert identifier links in Soar's WM, enabling the user to create a wanted initial structure on the WM's `^inputlink`.

One problem is that, to insert a link, a pointer to the parent link needs to be specified. But since initial structure in Soar's WM is not created until the `HLAAgent` object's `input`-function is invoked the first time (first time the Soar Input Phase is reached), such a pointer cannot be given by the time `insert_soar_link/2` is called. The problem was circumvented by creating a type, `IdLink` (of type long int), to represent the pointer. The `^inputlink` got number 0. Further insertions results in new unique `IdLink` numbers, created in an incremental fashion. Information about an insertion is stored in an object, `IdLinkInfo`. All `IdLinkInfo` objects are added to a table, `id_link_table`, to be used later when the actual insertion to Soar's WM can be made.

Arguments: - Parent identifier link (of type `IdLink`) of the inserted identifier link.
- The name of the inserted identifier link. (The name is used in a Soar program to access the identifier link.)

Returns: An `IdLink` (representing the inserted link) which can be used if further insertions are to be made.

register_soar_input/6
The function registers an association between a HLA attribute and a constant link in Soar's WM. Direction: RTI => Soar.

Arguments: - Name of the HLA object class the HLA attribute belongs to.
- The parent class-link (represented as an `IdLink`) an instance-link (representing an instance of the corresponding HLA object class) is attached to.
- Name of the instance-link representing an object instance the HLA attribute belongs to. (The name is used in a Soar program to access the identifier link.)
- Name of the HLA attribute
- A type (string, float, or int) the HLA attribute should be treated as.
- Name of the constant link representing the HLA attribute. (The name is used in a Soar program to access the link.)

Returns: `void`

register_soar_input_instance_unique/6
This function is analogous to `register_soar_input/6`. The difference is that the number of discovered instances of the corresponding HLA object class is restricted to one. If a second instance of this HLA object class is discovered, the instance is ignored. The meaning and number of arguments to this function is the same as in `register_soar_input/6`.

Arguments: See `register_soar_input/6`

Returns: `void`

register_soar_output/4
The function registers an association between a Soar output command and a HLA attribute. Direction: Soar => RTI.

Arguments: - Name of the "output-command-link". (The name is used in a Soar program to issue the output command.)
- Name of the HLA object class the HLA attribute belongs to.
- Name of the HLA attribute.

- A type (string, float, or int) the HLA attribute should be treated as.

Returns: `void`

run_execution/0
Starts the whole thing going. The function starts with initial preparations to the RTI (like subscribing, publishing, registering object instances, etc.), and then runs in the RTI cycle with the following order:
1) RTI Input Phase
2) Soar Input Phase
3) Soar Output Phase
4) RTI Output Phase

Arguments: None

Returns: `void`

7.7 Data Structures

7.7.1 Data Structures: RTI => Soar

Concerning data transfer from the RTI directed to Soar's WM, the SoHlCo system uses 8 tables:
`id_link_table`[1]
`HLA_class_info_table`
`HLAInstance_link_absent_table`
`HLAInstance_link_present_table`
`modified_values_table`
`non_modified_values_table`
`request_attribute_values_table`
`removed_HLA_instances_table`

HLA_class_info_table

Meta-information about associations between HLA objects (together with their attributes being subscribed to and that are destined to Soar's WM) and structure in Soar's WM needs to be stored so that later functions can operate properly. For example, when getting handles to HLA object classes and HLA attributes the names of the classes and attributes are needed. When subscribing to HLA attributes, handles to HLA object classes and HLA attributes are needed. HLA object class handles is also used to define what object class an object instance belongs to when it is discovered. Essential information concerning Soar's WM structure needs to be stored. For instance, a pointer to an `IdElement` object is used when an object instance of a certain object class is discovered, to insert a new identifier link representing the instance.

[1]`id_link_table` has already been described in the description of `insert_soar_link/2` in section 7.6 "User Functions"

Meta-information about a HLA object class with HLA attributes being subscribed to, together with the association with Soar's WM structure, is stored in an object of the `HLAClassInfo` class.

Working on the information given in the user function `register_soar_input/6` (or `register_soar_input_instance_unique/6`), `HLAClassInfo` objects are created and stored in a table, `HLA_class_info_table`. This table is accessed later when, from the RTI, new object instances are discovered or when new attribute values are reflected.

HLAInstance_link_absent_table and **HLAInstance_link_present_table**

These tables contain the information about present object instances together with their attribute values. All discovered HLA object instances are represented as objects of the `HLAInstance` class. A `HLAInstance` object also contains a table of attribute values previously reflected. Depending on how many attribute values have been reflected from the RTI for the first time, this table may not comprise the full set of attribute values expected to be reflected during the course of the federation execution. (Initially it is empty.)

Each `HLAInstance` object contains a pointer to an `IdElement` object representing the object instance in Soar's WM. This object is not created until the Soar Input Phase is reached after the instance has been discovered. One of the tasks in Soar Input Phase is to create the identifier links of HLA object instances that have not yet been created. To avoid having to go through all HLAInstance objects to search for objects with non-created identifier links, initially when a `HLAInstance` object is created it is placed in the `HLAInstance_link_absent_table`. When the function responsible for creating the identifier links in Soar Input Phase is invoked, it only needs to go through this table, and when an identifier link has been created, places the corresponding HLAInstance objects in `HLAInstance_link_present_table`. Once a `HLAInstance` object has been transferred to the `HLAInstance_link_present_table`, it will stay there for the rest of its lifetime.

---

modified_values_table and **non_modified_values_table**

In Soar Input Phase, newly reflected attribute values are updated in Soar's WM. It was considered unnecessary to have to go through all `HLAInstance` objects (and their internal tables with attribute values) in order to conclude which attribute values are newly reflected and hence should be updated in Soar's WM. This could result in an extensive search, especially if a considerable amount of HLA object instances are present in the federation.

For this reason, a "shortcut" object, `SOAR_HLAAttributeLink`, was designed. There is one `SOAR_HLAAttributeLink` object for each HLA instance attribute. A `SOAR_HLAAttributeLink` object contains a HLA instance handle and a HLA attribute handle (to uniquely define the instance attribute), and a pointer to an attribute value residing in some table in some `HLAInstance` object.

`SOAR_HLAAttributeLink` objects reside in the tables `modified_values_table` and `non_modified_values_table`. The objects in `modified_values_table` represent newly reflected attribute values that should be updated in Soar's WM. The objects in `non_modified_values_table` represent "old" attribute values that need not to be updated in Soar's WM.

The function in Soar Input Phase responsible for updating newly reflected attribute values in Soar's WM needs only to go through `SOAR_HLAAttributeLink` objects in `modified_values_table`, to update the corresponding value in Soar's WM, and then to place the `SOAR_HLAAttributeLink` objects in `non_modified_values_table`.

---

request_attribute_values_table

Each time a new HLA object instance is discovered, the corresponding instance handle is inserted in the table `request_attribute_values_table`. This table is inspected in the RTI Output Phase when calls that request attribute value updates are made.

---

removed_HLA_instances_table

The table `removed_HLA_instances_table` contains instance handles of HLA object instances that has recently been removed from the federation.

7.7.2 Data Structures: Soar => RTI

Concerning data transfer from Soar to the RTI, SoHlCo uses two tables:
`to_RTI_table`
`not_to_RTI_table`

The association between a Soar output command value and a HLA instance attribute (together with the value itself) is represented as an object of the `SoarOutput` class. There is one `SoarOutput` object for each association.

SoarOutput objects are stored in the two tables `to_RTI_table` and `not_to_RTI_table`. `to_RTI_table` contains `SoarOutput` objects representing Soar output values that should be updated on the RTI, *or* Soar output values that has previously been updated, but that the RTI has asked SoHlCo to provide an attribute value update of. `not_to_RTI_table` contains `SoarOutput` objects representing Soar output values that has previously been updated on the RTI, and need not to be updated again.

7.8 Implementation of Data Transfer

The following two subsections describe how data transfer between Soar and the RTI is implemented in the SoHlCo system. To make the reading easier, each subsection has been divided into the four phases constituting the main control flow in SoHlCo: RTI Input Phase, Soar Input Phase, Soar Output Phase and RTI Output Phase.

7.8.1 Implementation of Data Transfer: RTI => Soar

RTI Input Phase
Data from the RTI arrives as callback functions at any time during execution of the system. The callback functions are taken care of in the RTI Input Phase. Concerning data from RTI directed to Soar, there are three callback functions dealt with:

Discovery of a new object instance.
Taken care of in the function `RTIin_add_new_instance`.
The function performs 2 tasks:
- Create a `HLAInstance` object to represent the HLA object instance and place it in `HLAInstance_link_absent_table`. The next time Soar Input Phase is reached, an identifier link representing the instance will be created, and the `HLAInstance` object will be moved to `HLAInstance_link_present_table`.
- Add the handle of the object instance to `request_attribute_values_table`. The next time RTI Output Phase is reached, calls to request attribute values belonging to these instances are made to the RTI.

Reflecting attribute values.
Taken care of in the function `RTIin_update_HLA_attribute_values`.
Task:
- Each attribute value being reflected is decoded (from a byte-stream to a string, float, or an int) and updated in its proper table in the correct `HLAInstance` object. The corresponding `SOAR_HLAAttributeLink` object (if no such object exists — it is the first time this instance attribute is reflected — it is created) is placed in `modified_values_table`. The next time Soar Input Phase is reached, the newly reflected attribute values will be updated in Soar's WM (or a constant link representing the attribute value created, if it is the first time this instance attribute is reflected).

Removing of an object instance.
Taken care of in the function `RTIin_remove_HLA_instance`.
Task:
- The handle to the HLA object instance being removed is added to `removed_HLA_instances_table`. The next time Soar Input Phase is reached, all data connected to the removed instance will be deleted from the data structures in the `SoHlCo` object and deleted in Soar's WM.

Soar Input Phase
Data directed to Soar's WM is handled in Soar Input Phase. Soar Input Phase is implemented in the `HLAAgent`'s `input` function. `input` differentiates between if it is the first time it is invoked or if it is a subsequent invocation.

Concerning data transfer from the RTI to Soar, the first invocation of `input` involves two initial preparations:
- Inserts initial structure on the `^inputlink` of Soar's WM. Initial structure has been specified by the user via the function `insert_soar_link/2`, and the information resides in `id_link_table`. A function `SOARin_insert_original_links` inserts the identifier links in Soar's WM, and saves the pointers to the

corresponding `IdElement` objects.

- Once the pointers to `IdElement` objects have been retrieved, they need to be set in `HLAClassInfo` objects in `HLA_class_info_table`. (Previously these objects only held the "temporary" `IdLink` type to represent the identifier link.) A function, `SOARin_set_id_links_in_HLA_class_info`, performs this task.

Subsequent invocations of `input` involves three tasks:

- Identifier links in Soar's WM representing HLA object instances that have been removed from the federation are deleted. Furthermore, all information in the `SoHlCo` object concerning the object instances is removed. This information includes `HLAInstance` objects in `HLAInstance_link_absent_table` or `HLAInstance_link_present_table`, and `SOAR_HLAAttributeLink` objects in `modified_values_table` or `non_modified_values_table`. Handles to object instances that have been removed reside in `removed_HLA_instances_table`. The task is performed by the function `SOARin_remove_identifier_links`.
- HLA object instances that have just recently been discovered from the RTI do not have corresponding identifier links representing the instance in Soar's WM yet. `HLAInstance` objects representing such instances reside in `HLAInstance_link_absent_table`. An identifier link (together with an "ID-constant-link" identifying the instance inside a Soar program) for each such object instance is created in Soar's WM and all `HLAInstance` objects are then placed in `HLAInstance_link_present_table`. The task is performed by the function `SOARin_create_identifier_links`.
- Newly reflected attribute values are updated in Soar's WM. All newly reflected attribute values are found via `SOAR_HLAAttributeLink` objects residing in `modified_values_table`. The value of the constant link representing the attribute value is either updated or created (in case it has not been created yet) with the current value. All `SOAR_HLAAttributeLink` objects are then placed in `non_modified_values_table`. The task is performed by the function `SOARin_update_attribute_values`.

Soar Output Phase
The Soar Output Phase is not affected regarding data transfer from RTI to Soar.

RTI Output Phase
Concerning data from RTI directed to Soar, there is one task to be performed in the RTI Output Phase. That is to request an update on attribute values belonging to newly discovered object instances (and that has not already been reflected). Handles to those instances reside in the `request_attribute_values_table`. The function `RTIout_request_attribute_values` performs this task.

7.8.2 Implementation of Data Transfer: Soar => RTI

Soar Input Phase
Concerning data transfer from Soar to the RTI, the first invocation of `HLAAgent`'s `input` involves one initial preparation:

An `OutputManager` of the Agent Interface is created and registrations between all Soar output commands to be used and their corresponding functions in `SoarOutput` objects are created. These registrations are made in order to prepare for when, later in Soar Output Phase, Soar output commands trigger the right functions in corresponding `SoarOutput` objects. The task is performed by the function `SOARin_initiate_soar_outputs`.

Soar Output Phase
Soar Output Phase is implemented in `HLAAgent`'s `output` function. It executes the `OutputManager`. The result is that for every issued Soar output command, a corresponding `SoarOutput` object's `update` function is invoked, which updates the attribute value in the `SoarOutput` object and sets a flag showing that this object has an attribute value that should be updated on the RTI.

Furthermore, a function, `SOARout_move_SoarOutputs`, is called. By checking the flag in each `SoarOutput` object, it makes sure that the right objects are moved to the `to_RTI_table`. Later, in the RTI Output Phase, each object present in `to_RTI_table` will result in an update of HLA attribute values on the RTI.

RTI Output Phase
Attribute values that should be updated on the RTI are represented as `SoarOutput` objects residing in `to_RTI_table`. All of these attribute values are encoded (to a byte-stream from a string, float or an int), and updated via calls to the RTI. The `SoarOutput` objects are then placed in the `not_to_RTI_table`. This task is performed by the function `RTIout_update_attribute_values`.

RTI Input Phase
Concerning data from Soar directed to the RTI, there is one callback function the system deals with:

The RTI asks SoHlCo to provide an update of attribute values some other federate has requested an update of. By placing all such corresponding `SoarOutput` objects in `to_RTI_table`, it is guaranteed that the attribute value will be updated the next time RTI Output Phase is reached, even if it is an "old" value previously updated. If no such value yet exists (Soar has yet to send out this output value), the action is simply ignored. The attribute value will then be updated later when Soar sends out the output value for the first time. This task is performed by the function `RTIin_handle_provide_attributes`.

7.9 Converting Data from HLA Representation to Valid Soar Types and Vice Versa

The data structures in the `SoHlCo` object that store HLA attribute values do so in the "AgentInterface-representation", i.e. as strings, floats or ints. This means that HLA attribute values received from the RTI destined to Soar's WM are decoded (to a string, float or int) as soon as they have been reflected from the RTI, and HLA attribute values received as Soar output commands destined to the RTI are encoded (from a string, float or int) just before they are updated on the RTI.

In the case of transferring data from the RTI to Soar, it would, in fact, be slightly more efficient to wait with decoding the value until it is inserted in Soar's WM. This is due to the fact that in case an attribute value is reflected more than once in the RTI Input Phase before the next Soar Input Phase is reached, only the last value will be inserted in Soar's WM. Any previous reflections will then only result in unnecessary decoding. However, it was considered that the decoding functions did not require too much computational work (together with the fact that reflecting the same attribute value more than once in the same RTI Input Phase would probably be a rare case) to be of importance.

There are three functions for decoding a HLA attribute value from a byte-stream (to either a string, a float, or an int), and three functions for encoding a HLA attribute value to a byte-stream (from either a string, a float, or an int). Depending on how the federation to be participated in is designed, a user may have to manipulate some or all of these six functions in order to suit his/her needs in the case of encoding/decoding HLA attribute values.

7.10 Summary of the System

What follows is a summary of how the SoHlCo system works.

User functions
(- Constructor of `SoHlCo` object)
- `insert_soar_link/2`
- `register_soar_input/6`
- `register_soar_input_instance_unique/6`
- `register_soar_output/4`
- `run_execution/0`

Initial calls to the RTI
- Creates a federation execution (if not already created)
- Joins federation execution
- Gets object class handles
- Gets attribute handles
- Publishes object class attributes
- Subscribes to object class attributes
- Registers object instances

RTI Input Phase
- Discover Object Instance:
  Handled by `RTIin_add_new_instance`
- Reflect Attribute Values:
  Handled by `RTIin_update_HLA_attribute_values`
- Remove Object Instance:
  Handled by `RTIin_remove_HLA_instance`
- Provide Attribute Value Update:
  Handled by `RTIin_handle_provide_attributes`

Soar Input Phase (First invocation)
- `SOARin_insert_original_links`
- `SOARin_set_id_links_in_HLA_class_info`
- `SOARin_initiate_soar_outputs`

Soar Input Phase (Subsequent invocations)
- `SOARin_remove_identifier_links`
- `SOARin_create_identifier_links`
- `SOARin_update_attribute_values`

Soar Output Phase
- Execute `OutputManager`
- `SOARout_move_SoarOutputs`

RTI Output Phase
- `RTIout_update_attribute_values`
- `RTIout_request_attribute_values`

7.11 Some of the Problems Encountered

These are some of the problems or issues that came into light while implementing the SoHlCo system:

- **Inserting initial structure on Soar's WM's ^inputlink.**
  To be able to insert an identifier link on a parent identifier link, a pointer to the parent identifier link has to be given as argument. Since initial structure is not created until the `HLAAgent`'s `input` function is invoked the first time, these pointers cannot be retrieved by the time the user function `insert_soar_link/2` is invoked by the user. By letting `insert_soar_link/2` return a type `IdLink` (a long int) to represent a temporary pointer, so that the wanted structure could be saved to be inserted in Soar's WM later when possibility was given, the problem was circumvented.

- **Discovered HLA object instances have no initial pointer to an identifier link in Soar's WM.**
  If a HLA attribute value belonging to a just recently discovered instance is reflected, there is no place to insert the value in the next Soar Input Phase. Included in Soar Input Phase was a function, `SOARin_create_identifier_links` that before updates to Soar's WM are made, creates the identifier links that have not been created yet. `HLAInstance` objects representing

object instances that not yet have corresponding identifier links in Soar's WM, reside in a special table, `HLAInstance_link_absent_table`, so that `SOARin_create_identifier_links` does not have to search through all `HLAInstance` objects to check which objects to deal with, something that could be an extensive task if many object instances are present in the federation.

- **Unnecessary search of newly reflected HLA attribute values.** In Soar Input Phase, recently reflected attribute values are inserted in Soar's WM. The attribute values (both "new" and "old") are situated in tables belonging to `HLAInstance` objects residing in either `HLAInstance_link_present_table` or `HLAInstance_link_absent_table`. The set of newly reflected attribute values may be "scattered" all over these two tables. To search through all attribute values in order to decide which ones to insert in Soar's WM was considered a too extensive task, especially since a lot of HLA object instances present in the federation would result in a lot of attribute values to go through. By creating a "shortcut" object, `SOAR_HLAAttributeLink`, to represent each instance attribute, and by placing these objects in one of two tables (`modified_values_table` or `non_modified_values_table`), the amount of work to make the search was substantially reduced.

Chapter 8

Results

The SoHlCo system was tested in a federation together with two test federates. The test federates were, depending on what was tested for, registering a number of object instances, updating attribute values, reflecting attribute values (that the SoHlCo federate was updating), and removing object instances at random. Furthermore the test federates were following good advice by requesting attribute values as soon as an object instance was discovered (registered by the SoHlCo federate) and updating attribute values if they were asked by the RTI to provide attribute value updates (even if these updates previously had been issued).

The SoHlCo system was particularly tested for:

- Reflecting attribute values resulting in the values being inserted on the input structure on Soar's `^inputlink`.
- Updating attribute values in response to Soar issuing an output command.
- The act of updating attribute values in immediate response to reflecting an attribute value. (Soar rules were written especially to test for this.)
- Requesting attribute value updates on attributes belonging to recently discovered object instances.
- Updating attribute values that the RTI has asked the federate to provide an update on (even if these updates previously had been issued).
- Deleting all information about an object instance (together with the information about corresponding attributes) both in data structures and in Soar's WM if the instance was removed in the federation.

Concerning these topics the results were satisfactory. However, the one glitch was that occasionally the system crashed when an object instance were being removed from the federation. This often happened when the federation execution had been running for awhile and object instances had been registered and deleted by the test federates. When the crash occurred, it was always in response to an object instance removal. Due to time pressure, there was no possibility to localize the cause of the crash, but it was suspected that it was connected with memory leaks in Agent Interface. However, this could never be verified.

What is left to implement in the SoHlCo system is to replace the vectors with some better type of data structure. As it is, the use of vectors as data structures is not very efficient and functions dealing with the data structures tend to be more complex and less understandable than what they could be if for instance hash tables were used.

Chapter 9

Discussion

The SoHlCo system created is a connection between Soar and the RTI that transfers reflected HLA attribute values to Soar's WM, and that transfers Soar output commands to the RTI by updating HLA attribute values. Whenever a previously discovered HLA object instance is deleted from a federation, all information about the instance is removed from Soar's WM. As a federate, SoHlCo is not involved in Time management. It solves the initial value problem of HLA attributes belonging to newly discovered object instances by using advice from the RTI. Whenever a HLA object instance is discovered, SoHlCo requests from the RTI value updates on the attributes belonging to the instance. Likewise, whenever the RTI asks SoHlCo to provide updates on attribute values, SoHlCo follows good advice and updates the attribute values in question, even though these updates have previously been issued.

9.1 Including Time Management

The SoHlCo system does not contain the mechanisms to include Time Management services. However, by designing the main control flow of the system in a cycle similar to that of a time-involved cycle, the work of adding the mechanisms of Time Management services to the system is more of a trivial nature. This is relevant both for making SoHlCo an event-stepped or time-stepped federate. The mechanisms can simply be "wrapped around" the code that controls the four phases (RTI Input Phase, Soar Input Phase, Soar Output Phase and RTI Output Phase) so that the phases are executed in a manner in accordance with Time Management. Achieving this would not be difficult. It would be the same procedure as that of designing *any* time-involved federate.

If Time Management is added, there might no longer be needed for the SoHlCo federate to ask the RTI for advice (requesting attribute values) to solve the problem with not receiving initial values of attributes belonging to recently discovered object instances. If the federation is designed in such a way that phased initialisation is used, the participating federates will synchronize their actions together, and the SoHlCo federate is guaranteed it will reflect all attribute values that are updated by the other federates.

Concerning Time Management and advice from the RTI, there are basically three different alternatives in the manner the SoHlCo federate can be used:

1. No involvement in Time Management. (This is how the current SoHlCo version operates.) To solve the initial value problem of attributes belonging to recently discovered HLA object instances, the federate needs to use advice from the RTI.
2. Involvement in Time Management, but no use of advice from the RTI. To be guaranteed that the initial value problem should not arise, the federation (that the SoHlCo federate is participating in) must be using phased initialisation.
3. Involvement in Time Management *and* using advice from the RTI. The federation may consist of both federates that are involved in Time Management and federates that are not. In this case the SoHlCo federate could be using advice from the RTI (as well as being time-involved) for the sole purpose of responding correctly to the calls of the non-time-involved federates.

This gives rise to different possibilities in how to design a SoHlCo system so that Time Management can be included. The most elegant solution would be if the user can specify which one of the three different alternatives mentioned above is desired, and the system would behave accordingly. The functionality dealing with advice from the RTI could easily be "shut off" if it is not needed. A thorough investigation and verification should however be conducted in this case, in order to determine if the system operates appropriately. Examination of the possibility of adapting SoHlCo for reusability purposes may make such verification a trifle difficult. An alternative is to construct three different versions of SoHlCo, one for each of the alternatives mentioned above. Verification of one such system would be easier, but it requires that the user picks the correct version to suit his/her needs.

Another choice to be made when including Time Management is to whether the SoHlCo federate should be time-stepped or event-driven. No further analysis has been made in order to determine exactly what the impact is to either make SoHlCo time-stepped or event-driven. Again, the construct could be conducted in different manners. A system could be designed that can take care of both the time-stepped and the event-driven case, guided by the user's choice. Alternatively, two versions could be constructed, one that operates time-stepped and one that operates event-driven.

9.2 Ownership Management

As mentioned in section 6.8 "Ownership Management" in the analysis part of this thesis, no extensive analysis work has been conducted to figure out how to integrate Ownership Management into the system. For the uses thought of when designing the SoHlCo program, there was no need to integrate any Ownership Management services.

If there is a desire to adapt SoHlCo to include Ownership Management services, an extensive analysis needs to be conducted in order to decide how exactly it should be done. Including such services will most likely also affect other parts of the system, so a re-verification that the system behaves correctly will have to be done. It is uncertain how including Ownership Management services will affect the issue of trying to adapt SoHlCo to making it reusable to other AI architectures than Soar.

9.3 Including Functionality Dealing with HLA Attributes Representing Complex Datatypes

The SoHlCo system was implemented with the assumption that the HLA attributes dealt with can easily be converted to simple datatypes like ints, floats, or strings. In a federation, a HLA attribute can represent more complex datatypes. Technically it should be possible to extend the SoHlCo system to also deal with such complex datatypes. The result would be that whenever a HLA attribute representing a complex datatype is reflected, the attribute is decoded, its composite parts separated and the several corresponding constant links on the `^inputlink` in Soar's WM are updated. Furthermore, whenever Soar issues several output commands representing a more complex datatype, the command values are together encoded to a HLA attribute and the attribute is updated in the federation.

It seems simple in theory. The task however seemed too complex and difficult that no analysis on how to solve the technical problems was carried out. The main problem is that a user's wishes on how to encode/decode a HLA attribute and also how to represent the complex datatype in Soar's WM may vary from one federation execution to the next. This creates a demand to make some kind of general ingenious solution that would enable the user to carry out his/her plans regardless of his/her desires.

Another effect of including complex datatypes is that a large part of the system would be affected, resulting in the requirement to conduct a thorough verification that the system behaves correctly. Trying to adapt the system for reusability purposes would make such a verification even more difficult. Unfortunately, when the work this thesis is based on was conducted, there was no time to carry out such a verification.

9.4 Interactions

The SoHlCo system deals with HLA objects (and their attributes), but not with HLA interactions. As mentioned in section 6.9 "Adding Interactions into the System" in the analysis part of this thesis, there are some semantic issues to deal with regarding representing an interaction in Soar's WM, but once these issues are settled the technical difficulties of including interactions in the SoHlCo system could easily be overcome. A structure dealing with HLA interactions similar to the already existing structure dealing with HLA objects and running in parallel with it, could easily be constructed. The main difference would be that an interaction representation can be taken away as soon as the meaning of it has been processed (following the definition of an interaction). As a result, the interaction structure would be simpler than the HLA object structure. For example, there would be no need for a "shortcut" object like the `SOAR_HLAAttributeLink` regarding HLA objects. The reason for the "shortcut" object was to reduce the time to search for newly updated HLA attribute values. With interactions, it is guaranteed that all interaction representations present in the system are "new" values and should be processed. Thus the need for a "shortcut" object is eliminated, making an interaction structure less complex than the HLA object structure.

9.5 Making the System Reusable for other AI Architectures than Soar

A highly interesting question is whether a constructed system could be adapted to be reusable for other AI architectures than Soar. Trying to seek an answer to this question was a task included in the work this thesis is based on. Unfortunately, during the time the work was conducted, no real yes-no answer could be deduced from it.

However, one obvious conclusion to be drawn is that, in order to adapt SoHlCo to be reusable, it is important to eliminate all Agent Interface-specific classes in the `SoHlCo` class. Recall from Figure 7.1 that the `SoHlCo` class is the "hub" of the SoHlCo system. The `SoHlCo` class contains the main functionality of the system, and is the actual connection between Soar and the RTI. As it is, the data structures residing in the `SoHlCo` class are pervaded by classes directly derived from Agent Interface. (These classes are mainly pointers to locations in Soar's WM structure.) For the `SoHlCo` class to be able to work in a more general way, these classes would have to be replaced by other more loosely defined classes. One suggestion is to use the template mechanism present in the C++ language for it to work.

The data structures used in the `SoHlCo` class are designed to fit with the WM structure in Soar. Soar's WM structure is characterized as being a non-strict hierarchy. Some other AI architecture may use a different structure, meaning that the data structures in SoHlCo may not be adequate for this AI architecture. With this in mind, it would be essential to work out a structure to save values in a way that can easily be used and tailored to suit any AI architecture in question. Possibly, the work of associating HLA attribute values with the internal structure of the AI architecture being used could be "lifted out" of the `SoHlCo` class and placed somewhere else, resulting in an extra (reusable) layer in the system. The work conducted for this thesis could not give an answer as to whether this is possible or not.

Functionality-wise, the way in which many functions in SoHlCo operates is highly influenced by Agent Interface. Some examples:

- Calls are made from the `HLAAgent` object *back* to the `SoHlCo` object. The `SoHlCo` object starts the Soar Input Phase and Soar Output Phase by making a call to the `HLAAgent` object. In turn, the `HLAAgent` object has to make several calls to the `SoHlCo` object when executing its `input` or `output` functions.
- Initial structure on Soar's WM (on the `^inputlink`) is created when the `HLAAgent` object is executing its `input` function for the first time. This circumstance has given rise to the creation and adaptation of a few functions in the `SoHlCo` object assigned to the task. An example is the creation of the `IdLink` datatype, a temporary aid to represent pointers to locations in the Soar WM structure.
- Soar output commands are registered with an `OutputManager` when the `HLAAgent` object is executing its `input` function for the first time. A function in the `SoHlCo` object has been created and assigned for the task.

These are all functionality characteristics whose presence is an immediate result of using Agent Interface as the interface to Soar. They would not be essential when using some other Soar interface or some other AI architecture. In order to be able to design and create a reusable system, initially a general functionality that can be used by any interface should be constructed. Figuring out a way to connect an AI architecture interface to this system would be a subsequent task. Using this approach may result in a necessity of having extra layers in the system.

Finally, it is the author's strong opinion that to create a reusable system, the key word is simplicity. Having a simple functionality and using simple data structure will make life much easier when verifying that the system behaves correctly. For example, to become more efficient, the SoHlCo system uses a "shortcut"-object, `SOAR_HLAAttributeLink`, situated in one of two extra tables. Only by adding this extra feature made it notifiably more difficult to get an easy look-over of the system. For this reason and others, the task of trying to figure out a way to make SoHlCo reusable was abandoned. The best approach when trying to create a reusable system it seems, is to create an as simple system as possible, trying to adapt it for reusability purposes, and then (if necessary) modifying the system to improve efficiency.

Chapter 10

Conclusions

10.1 Concluding remarks

As may be clear from this report, there are a lot of aspects to consider when dealing with HLA. A vast number of eligible options when it comes to design of federation, HLA attribute representation, Time Management, Ownership Management, interactions, etc., makes it difficult to create a general system that can suit every user's needs. The fact that the scopes of some of these areas overlap together with the task of dealing with all the many situations that may occur in a federation, would complicate the work even further. A system where a user can specify its desires by setting a number of variables could perhaps be constructed, but it would be a monster-job trying to glue all concepts together, resulting in a rather complex, hard-to-verify system.

However, if the circumstances are such that a specified behaviour of both (Soar) federate and federation exists, creating an automated connection between Soar and the RTI is a perfectly doable task. The drawback then is that the user is restricted to the decided behaviour of the implemented system.

When it comes to creating a reusable HLA connection system that can be utilized by many different AI architecture interfaces, it seems that the best approach is to start designing in the "RTI-end" of the system, with reusability factors in mind from the very beginning. The task of connecting AI architecture interfaces to this system could then be commenced, rest assure that a firm base design already exists. Trying to modify an already existing system to suit reusability needs, seems a bit too complicated. Too many functionality dependencies from the AI architecture interface being used may exist. At least, that was the case with the system created for this thesis.

Bibliography

[1] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall PTR, 2000.

[2] Judith Dahmann, Frederick Kuhl, Richard Weatherly. *Standards for Simulation: As Simple As Possible But Not Simpler The High Level Architecture For Simulation*. Simulation, vol. 71, no. 6, 378-387. 1998.

[3] Judith Dahmann, Richard Fujimoto, Richard Weatherly. *The DoD High Level Architecture: An Update*. In Proc. of the 1998 Winter Simulation Conference, Washington D.C., 797-804. 1998.

[4] HLA on Wikipedia. 2008.
URL: http://en.wikipedia.org/wiki/High_Level_Architecture_%28simulation%29

[5] Niklas Wallin. *Modelling Psycho-physiological effects in Computer Generated Forces*. Master Thesis report, FOI. 2002.

[6] John Laird, Paul Rosenbloom, Allen Newell. *Soar: An Architecture for General Intelligence*. Artificial Intelligence, 33: 1-64. 1987.

[7] Paul Rosenbloom, John Laird, Allen Newell. *The Soar Papers: Readings on Integrated Intelligence*. 1993.

[8] Soar on Wikipedia. 2008.
URL: http://en.wikipedia.org/wiki/Soar_%28cognitive_architecture%29

[9] Jill Fain Lehman, John Laird, Paul Rosenbloom. *A Gentle Introduction to Soar, An Architecture for Human Cognition: 2006 update*. National Science Foundation, Grant No. 0413013. 2006.
URL: http://ai.eecs.umich.edu/soar/sitemaker/docs/misc/GentleIntroduction-2006.pdf

[10] John E. Laird and Clare Bates Congdon. *The Soar User's Manual, Version 8.6.3*. The Regents of the university of Michigan. 2006.
URL: http://ai.eecs.umich.edu/soar/sitemaker/docs/manuals/Soar8Manual.pdf

[11] John E. Laird. *The Soar 8 Tutorial*. University of Michigan. 2006.
URL: http://ai.eecs.umich.edu/soar/sitemaker/docs/tutorial/TutorialPart1.pdf
URL: http://ai.eecs.umich.edu/soar/sitemaker/docs/tutorial/TutorialPart2.pdf
URL: http://ai.eecs.umich.edu/soar/sitemaker/docs/tutorial/TutorialPart3.pdf

[12] Allen Newell. *Unified Theories of Cognition*. Harvard University Press. 1990.

[13] Jan Skansholm. *C++ Direkt*. Studentlitteratur AB, 2000.