Graduate School of Information Technology and Mathematical Sciences
University of Ballarat

2009

AOP and HLA: A New Aspect on Distributed Simulation Development

Mr. Timothy J. Pokorny
University of Ballarat
t.pokorny@ballarat.edu.au

This thesis is submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

University of Ballarat
PO Box 663
University Drive, Mount Helen
Ballarat, Victoria, 3350
Australia

Submitted in February 2009

Abstract

Underpinning the development of distributed simulations in the defence community, the High Level Architecture (HLA) has gained acceptance due in part to its support for a broad level of interoperability. Encompassing a framework that loosely couples together simulation components developed and deployed on a diverse range of platforms, the HLA has the potential to enable increasing interoperation between otherwise disparate simulations.

Although it has long been used for simulation efforts in the defence domain, use of the HLA within the wider business community has thus far been minimal. In domains where a wide variety of proprietary, customised simulation tools and generic desktop applications alike are used for simulation purposes, use of the HLA can help enable increased reuse and interoperability. However, while capable of supporting such a goal, the HLA requires expert skills and training that do not exist in these domains.

Aspect Oriented Programming (AOP) methodologies partition the development of a software system into a number of separate "aspects". Some aspects relate to the core business logic of the application, while others relate to system-level facilities such as applications distribution (perhaps via the HLA). To form a complete application, a number of aspects are automatically woven together according to a set of weaving rules created by developers. While the final system represents a mixing of all aspects, the process of developing each one is conducted in isolation. This in turn allows developers to work without the need for an in-depth knowledge of the underlying technologies used by other components.

This thesis develops a method for combining AOP and HLA, leveraging the separation-of-concerns approach used by AOP to allow the creation of core models, free from simulation distribution semantics. Through the use of automated tools, these models are then woven with a generic-HLA aspect, producing an HLA-enabled simulation component. Using AOP in this manner removes the need for model developers to have an in-depth understanding of the HLA, helping to remove the prime factor restricting a broader uptake of distributed simulation technologies: development complexity.

Statement of Authorship

Except where explicit reference is made in the text of the thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis by which I have qualified for or been awarded another degree or diploma. No other person's work has been relied upon or used without due acknowledgement in the main text and bibliography of the thesis.

Signed: [illegible]	Signed: [illegible]
Dated: 9/12/09	Dated: 4/12/09
Mr. Timothy J. Pokorny	Dr. Philip A. Smith
Candidate	Principal Supervisor

Acknowledgements

Although I ultimately stand alone when it comes to the responsibility for this work, the number of people involved some way or another in its production is considerable. I've been toiling away at this for many years and through this time many people have influenced and contributed to the process. Whether it be materially or less tangibly, the value of these endowments has always been welcomed and very much appreciated. Accordingly I would like to take a brief moment to specifically thank a small group of people who have been a consistent presence throughout my time as a student and without whose support and help I would never have been able to finish.

Firstly, I have to thank the academic and support staff at the University of Ballarat. In particular I must thank *Professor Sidney Morris* for instilling in me the idea that if I can't explain what I'm doing in two sentences then I don't know what I'm doing at all. I would also like to mention my gratitude to *Di Clingin* for helping to ensure that I remained funded and for responding to any requests with speed and care.

Throughout my time as a PhD student I've been fortunate enough to have a small group of people who have been willing to listen to my often incoherent ramblings or provide a welcome distraction. *Josh Stewart, Sae Ra Germaine* and especially *Cameron Tudball* have always provided this companionship, something especially needed when your work from the solitary confinement of a home office. Providing the same relief but in a more topical sense were *Dr. David Andrews* and the soon to be Dr's, *Lance Burns* and *Anthony Cramp*. Whether it be through late night marathon games of pool or acting as a general sounding board for ideas, frustrations and insights, their support has been invaluable. Thank-you all.

I have known *Michael Fraser* since the beginning of my PhD, and from that time I've had the privilege of calling him one of my closest friends. We've worked together on numerous projects, from insanely bad lego-based stop motion animation to sugar fuelled plans to take over the world with Open Source Software. He is someone I implicitly and unquestionably trust in the trenches and the projects that have been the biggest success in my life thus far all bear his influence. Thank-you.

My supervisors, *Dr. Philip Smith* and *Dr. David Stratton* deserve special mention. We have all been working together on various projects for over 6 years now and I can't think of two people who have been more influential on my professional development than them.

Their attention, feedback, encouragement and guidance throughout this entire period has been indispensable and I cannot thank them enough.

Until almost the end of my time as a PhD, *Marni Ryan* and I were partners. The love and support I received from her during this time was altogether above and beyond anything I could ever expect; offered unconditionally and without reservation. As the years ticked by on my thesis she never nagged, never pushed, never pressured me to finish. I will always consider her a part of my most immediate family, and can only hope that one day I can somehow repay to her that which she sacrificed for me.

Finally, I must reserve me most heartfelt thanks to my parents, *Stan and Elizabeth Pokorny*. It is difficult to adequately articulate the full effect my parents have had on me. Although I would not have believed it when I started as a student, over time I can see more and more of their influence and traits coming to the surface in the way I think about and approach life in general. They put me in front of a computer and despite numerous hurdles provided me with a stable, worry free environment in which I could explore and learn. Everything I needed they provided and without their support I would never have finished.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ACM	Aircraft Manager Federate
AOP	Aspect-Oriented Programming
API	Application Program Interface
ASL	Action Semantics Language
ATC	Air Traffic Control Federate
ATO	Air Transport Operations Federation
DLC	Dynamic Link Compatibility
DMSO	Defense Modeling and Simulation Office
FM	Flight Manager Federate
FOM	Federation Object Model
HLA	High Level Architecture
IEEE	Institute of Electrical and Electronics Engineers
MDA	Model Driven Architecture
OCL	Object Constraint Language
OMG	Object Management Group
OMT	Object Management Template
OO	Object Oriented
OOP	Object-Oriented Programming
OSS	Open Source Software
PIM	Platform Independent Module
PSM	Platform Specific Module
RTI	Run-Time Infrastructure
SISO	Simulation Interoperability Standards Organization
SOM	Simulation Object Model
UML	Unified Modelling Language
UoB	University of Ballarat
WBC	Wider Business Community
XMI	eXtensible Metadata Interchange

Chapter 1

Introduction

Distributed simulation is recognised as a useful and important part of training, scientific modelling and acquisitions. The ability to assess the effectiveness of a new product design, or financial model without the expense and uncertainty of implementing it allows for the identification of defects or weaknesses at a stage when they are easily rectified. Involving numerous simulation components connected over a network, the standardised High Level Architecture (HLA) is widely recognised as the primary distributed simulation facilitator within the defence community.

Encompassing a framework that loosely couples together simulation components developed and deployed on a diverse range of platforms, the HLA has the potential to enable increasing interoperation between otherwise disparate simulations. While the benefits of distributed simulation are well understood and practised in the defence domain, uptake of the associated technologies in the wider business community has thus far been minimal.

Despite comparatively minimal application of distributed simulation beyond the field of defence, simulation in general remains a vital component in many enterprises. General productivity tools (such as spreadsheets) and customised simulation applications tend to be favoured over distributed technologies, primarily due to their ease of use and pervasive nature. While customised simulation tools focus on abstracting the problems of a particular domain, generalised desktop tools are broadly available, well understood and simple to use. In such an environment, the relative complexity of distributed simulation technologies appears to provide little benefit when considering the expert training and skills required.

However, while the tools typically employed within the wider business community offer a simplified development experience for many simulation problems, significant reuse, interoperability, design and scalability issues exist. Designed with such issues in mind, the HLA has the potential to address these shortcomings and support both expanded interoperability and increased model complexity.

While capable of delivering many benefits, the significant costs and complexities involved in the development of HLA simulations have thus far constrained its broader uptake. To help address this problem, the research presented here seeks to abstract the HLA from the model development process.

Aspect-Oriented Programming (AOP) builds on traditional Object-Oriented (OO) software development approaches and defines a process for modularising and separating multiple concerns during software development. Where traditional methods require the tangling of system-level details (such as the HLA) within the business logic of a software system, AOP introduces a new unit of modularisation that allows their development to be quarantined. Isolating such concerns allows the majority of a system to be developed without the specialist knowledge implementation of these system details necessitates. Such an environment has clear potential benefits in application of the HLA within the wider business community. However, AOP is only a facilitator, allowing components to be developed separately. Considerable specialist knowledge is still required to create the HLA portion of a system.

Leveraging AOP as a mechanism for separating simulation model development from the low-level details of the HLA, this research discusses methods that can be used to automatically extract HLA semantics from a pure Object-Oriented (OO) model. This information can then be used to manipulate a generic HLA component, removing the need for the development of a custom solution (and the specialised knowledge such an effort would entail). Taken together, such an environment would allow for the abstraction of the HLA from the simulation development process.

This research discusses methods that allow generic, non-distributed models (represented as plain OO code) to be automatically rendered as HLA simulation components. Removing the requirement for expert HLA skills and training, such a facility would significantly simplify the development of new distributed models and help expose existing models to HLA-based distributed simulation and the benefits it brings. This work focuses on the questions that arise when attempting to achieve this goal.

1.1 Background

Used as a decision support mechanism for many years, simulation involves the investigation and assessment of the various effects and outcomes of a model given particular inputs and events. For example, a financial services corporation may use a stock market model to ascertain the likely behaviour of the market should interest rates increase. Alternatively, a product manager may produce a financial model in order to determine the effects on productivity and profitability should five new employees be hired.

In pursuit of developing these simulation models, two categories of tools are generally used; generic desktop productivity applications and specialised simulation tools. Each of these tools offers both a number of advantages and disadvantages.

Spreadsheet Applications

The most popular and prevalent tool used for simulation purposes within the wider business community is the spreadsheet. Spreadsheet applications provide a general-purpose environment that caters well to a larger number of diverse simulation problems. Inbuilt support for common mathematical functions is enhanced by facilities that allow the more complex logic associated with simulation models to be developed via traditional programming languages and plugged in as modules.

Combining excellent support for the development of small numerical models with facilities for formatting and presenting the results enables solutions to many small simulation-like problems to be developed in a rapid fashion. However, perhaps the most compelling of all advantages this class of tool presents is their desktop commodity status. General-purpose tools such as spreadsheets are widely available and relatively inexpensive. Their support for a large breadth of purposes has seen them become the de-facto standard for simulation activities in the wider business community.

However, while spreadsheet applications provide an environment well suited to the development of smaller solutions, as the simulation models involved begin to grow, the limits of these desktop tools are quickly reached.

Spreadsheets depend heavily on the location of information. The use of data is inherently tied to its position in the spreadsheet, meaning the functions intended to operate on this data are extremely sensitive to even the slightest repositioning. A small change can trigger a multitude of errors in any moderately sized spreadsheet. One of the most commonly noted problems with spreadsheet applications is the prevalence of errors. A review of the literature presented later in this document highlights the high occurrence of error observed in spreadsheets, many citing the strict dependence on the location of information as a prime cause. As a consequence of this condition, the maintenance costs of spreadsheets increase and the reliability of the results they produce is reduced.

Strict dependence on the location of information causes model composition and reuse to suffer greatly. The ability to create a new entity entirely from pre-existing components is a powerful concept and one that has the potential to reduce the costs involved in composing simulation models. The ability to then reuse these entities in other models generates additional savings and increases the value of such components. However, the rigid structure of spreadsheets does little to support this type of composition and reuse, consequently resulting in increased initial development and ongoing maintenance costs. Additionally, the generalised interface is unable to express or capture the detailed logic often associated with simulation development, forcing developers to use traditional programming language facilities. Seeking to address some of these shortcomings, more specialised, simulation oriented tools have also become popular for model development.

Specialised Simulation Tools

While spreadsheet applications are the most pervasive tool used for simulation purposes within the wider business community, specialised simulation tools often provide a more compelling model development environment. Unlike spreadsheet applications, specialised tools are designed specifically to address the problems of simulation model design and construction. Where spreadsheets deal in generic terms, a custom tool is able to provide users with a comfortable setting that enhances their ability to develop such simulation models. Removing the major drawbacks associated with spreadsheet based simulation development, domain specific applications allow users to work in a setting designed specifically to meet their needs.

Free from the grid-based interface of spreadsheets, specialised tools are able to support simplified methods of model development. Often graphical in nature, these tools allow for greater comprehension and understanding of a simulation model and provide additional support for novice users lacking an in-depth knowledge of simulation. Such facilities help reduce initial development time and simplify the maintenance process. As with spreadsheet applications, where the required complexity cannot be supported by the interface, these tools also allow a developer to utilise traditional programming tools and insert the additional modules into the model. When combined, these capabilities result is an environment that scales well from small and simple models to larger and more complex ones.

However, despite offering many improvements over spreadsheet applications, specialised simulation tools present a large number of serious shortcomings that hamper their usefulness in many situations.

Specialised, graphical interfaces allow for greater human comprehension of a model and allow even novice users to create simulations. While the benefits are clear, a certain level of training is required to learn the customised language of the application. Further, such knowledge is not portable from one tool to another. Although the domain specific nature of many specialised tools provides an environment in which domain experts can be productive and work efficiently, generic tools such as spreadsheets can be useful in much larger array of situations. The specialised nature of such tools restricts their usefulness to the situations they were designed for.

Despite the shortcomings identified above, the biggest obstacle specialised simulation tools present is their proprietary nature. Although many tools support the reuse of simulation models or model components that were developed in the same tool, any attempt to leverage this investment in another environment is removed. While this restriction is acceptable in a number of circumstances, in situations where models from a variety of domains and specialities must be brought together, the lack of interoperability support in these tools obscures any development benefits they may impart.

1.2 Motivation

While each type of tool used for simulation in the wider business community presents advantages and disadvantages, a common shortcoming among all is a lack of interoperability. In a setting where numerous differing tools are all used for similar purposes, the ability to leverage investments made in the development of simulation models, independent of those tools, is severely restricted.

All simulations revolve around data. In any given model, regardless of tool or environment, information is created, updated and removed, according to its programmed behaviour. The primary goal of distributed simulation is to share the information produced by one simulation, with a number of other simulations. This allows individual simulations to take advantage of the data produced by others, without needing to implement the logic required to produce it. In this way, the burden of performing the simulation is shared.

The standalone simulation models typical of those produced by the desktop tools common to the wider business community do not have this ability. Without facilities to share information about the creation, alteration and removal of data, standalone simulations cannot interoperate with one another. This in turn restricts the size and complexity of such simulations, in addition to their reuse value. An excellent model for predicting stock market fluctuations in the face of interest rate changes is of little use when attempting to consider these effects in a broader context unless that model can work with other such excellent models that focus on alternate facets of the subject.

Ideally, simulation models could be built using whichever environment best suited the task at hand. They could then brought together and used as part of a larger model, regardless of tool or platform. Designed to address this issue, use of the HLA could help realise such a goal. Providing a low level infrastructure, use of the HLA to link together otherwise standalone models would provide a greater return on the investment made in their development.

While capable of addressing the common interoperability problem shared by simulation tools within the broader business community, use of the HLA is regarded as a complex and costly process. Notwithstanding the potential reuse and interoperability advantages it could provide, minimal application of the HLA beyond the defence community has shown the reluctance within mainstream domains to support a technology that requires expert distributed simulation knowledge and programming skills. As it currently stands, *the development costs and complexities of the HLA render it unsuitable within the wider business community.*

Much recent research has focused on reducing the development burden associated with the HLA and addressing its useability issues. In turn, this has lead to advancements that greatly reduce the time and effort involved in the production of HLA-based distributed simulations, making it more cost-effective and attractive. However, despite a bulk of research addressing the many problems, few researchers have investigated how the HLA can be removed entirely, thus making it suitable for the wider business community.

Some success has been gained in previous work that focused on attaching the HLA to specific specialised simulation tools through "tool specific interfaces". In that situation, certain tools were modified to add HLA support and allow their models to participate in distributed HLA simulations. This research is conceptually an extension of that effort. The focus in this work is on the development of a generically applicable solution. In this case, the specifics of a particular specialised tool cannot be considered in an effort to find a solution that has the potential to be applied within any tool. The use of AOP as a facilitator helps achieve this goal. AOP facilities exist for many platforms and environments and its concepts are based on consistent underlying theory. Through the use of AOP the potential to employ the developed solutions in a range of specialised simulation tools is maintained. It is against this background that the author draws motivation for this research.

1.3 Scope

The primary focus of this work is on the development of methods that would allow a generic simulation model to be rendered as a HLA distributed simulation component. However, before considering the significance of this goal, a clear definition of what constitutes a generic model is required.

While the landscape of tools and applications used to develop simulation models in the wider business community is broad and varied, at a low level, all models will at some point share a common form. Be it functionality pieced together via the grid-based interface of spreadsheet applications, or component behaviour composed via the drag-and-drop interfaces of specialised tools, at some point in a model's lifetime, it will become basic program code. Even if the portable format in which the model is stored is never directly translated into code itself, the execution environment that interprets that format will have been developed in some standard programming language. However, to support model execution, these tools make use of proprietary methods and frameworks. In turn, their use pollutes the model representation and creates non-generic dependencies.

For a model to be truly generic, it must contain only information about the system being represented and not the supporting infrastructure used to execute it. Remove the tool-specific information and only the "pure" model remains. It is the development of this business logic that defines the lasting value of a simulation model. In this research, we are referring to a generic model as one that consists of pure object-oriented code, free from any notion of application distribution of distributed simulation. While object-oriented programming skills can be viewed as "specialist," they are pervasive within the wider business community (unlike distributed simulation). Given this, we arrive at the following definition in the context of this research:

"A generic model can be described as a pure, object-oriented representation of a system that includes only the details salient to its operation and not the supporting execution infrastructure."

1.4 Contribution

Significant overlap exists between general OO programming theory and the HLA specification. Many of the facilities provided by the HLA are merely avenues through which information about the state of a simulation can be distributed to other simultaneously executing simulation components. As the state of some simulation object changes, this update can be reflected to other simulations, in turn allowing them to take some course of action based on the new information. In a standalone (or non-distributed) model, such events would still occur. However, the absence of any distribution framework would mean that these events are not shared with other simulations. For all intents and purposes, the pure model is monolithic.

Through the use of AOP and an investigation to the parallels between the HLA and general OO theory, this research develops methods that allows HLA semantic information to be extracted from pure models and renders them as fully distributed simulation components.

Such advancement in the state of the art would allow generic simulation models to be mapped automatically into components ready for use within the HLA, without mandating that simulation developers obtain direct knowledge of the HLA. This simplification would remove the primary barrier preventing a broader uptake of distributed simulation within the wider business community. Larger and more complex models could be developed, providing more reliable and in-depth results. Where previously the return on investment in a model was minimised due the inability to reuse models developed across differing tools, the interoperability benefits of the HLA can help to enhance them.

The complexities of distributed simulation have been well documented. All users (be they current or potential) stand to gain from a solution that lowers the barriers of entry and provides for increase focus on the core model issues rather than the development platform. Beyond the wider business community, areas in which the HLA is already used also stand to benefit. The reduction in development complexity and enhanced focus on model issues (rather than those of the HLA) help reduce development costs and time. Further, the simplified process aids in model comprehension and the identification of potential errors or weaknesses. Together, these benefits form a significant contribution and advancement in the current state of the art.

1.5 Overview

This thesis is arranged into 9 chapters. The first half, comprising chapters 2-5 lay out the motivation and background of this work. The second half describes the contribution of this work, presenting solutions to the identified problems and discusses experimental results.

Chapter 2 discusses the simulation tools used within the wider business community, their strengths and shortfalls. Chapter 3 introduces distributed simulation, and more specifically, the High Level Architecture, discussing how distributed simulation can be used to address the problems of tools used within the wider business community.

Chapter 4 highlights the problems that currently deem the HLA unsuitable for meeting the needs of commodity tools, primary among which is development complexity. This chapter highlights some of the technologies and approaches that have been identified within the distributed simulation community as potential answers to these problems, focusing particularly on the Model Driven Architecture (MDA) and Aspect-Oriented Programming (AOP), the latter of which is considered the most suitable for meeting the goals of this research.

Chapter 5 identifies where the gap exists in the current state of the art, and how AOP alone is not sufficient to solve the particular problem set motivating this work. It provides a set of research questions that capture the intent of this work and introduces the experimental framework that is used to assess the solutions raised in later chapters.

Chapters 6-8 form the bulk of contribution made by this work. They introduce a set of solutions aimed at addressing the research questions raised in Chapter 5 and present the results of experimentation.

Chapter 9 concludes this work, briefly identifying fertile areas for further research.

Chapter 2

Simulation in the Wider Business Community

Removing the complexities associated with distributed simulation development so that the wider business community can benefits from the advantages it can bring is the primary motivation for this research. This chapter discusses how simulation is used within the wider business community and what the associated problems are.

2.1 What is Simulation?

What is simulation and why is it useful? Before delving into an analysis of the tools that are commonly used within the wider business community for simulation purposes, some consideration must be given to the nature of simulation and the benefits it provides. As a starting point, the author Banks in [8] defines simulation as "an imitation of the operation of a real-world process of a system over time". While this definition does capture the essence of simulation (a replication of some process over time), it is somewhat restrictive to confine contemplation to real-world processes and systems.

From an economist attempting to predict the movement of foreign currency rates [47], to the physicist endeavouring to uncover the mysteries of dark matter, simulation can be found in many diverse situations. While the topic of currency rates fit clearly in Banks' definition, matters of a cosmological nature prove more challenging. In consideration of such situations, a broader definition is required. Perhaps an apt expansion of the previous definition would be: "An imitation of an environment or phenomenon over time, performed for the purpose of investigation, exploration, training or decision support".

Whichever form is preferred, simulation is leveraged to advantage in numerous and diverse fields. Given this situation, one is naturally drawn to pose the question, "what benefit or assistance does simulation provide?"

Why use Simulation?

Simulation is used for many purposes. When an airline pilot is trained, they are done so using a flight simulator. When an engineer wishes to explore the effects of stress on a bridge design or an environmental scientist wishes to investigate the effects of global warming on the average temperature of the planet, simulation is used. Where the direct observation of a live system is impractical [101] or dangerous, simulation can be used to assess or explore various outcomes, to ask various questions in a safe and cost effective manner.

The benefits of using simulation have been noted in many places with the following list compiled from [8, 9, 58, 103]:

- Time Compression and Expansion: Time can be compressed and expanded in order to more closely observe an event or observe occurrences that unfold over a large amount of time quickly.
- Understand "Why": Close analysing a system modelled by a simulation can assist in understanding why a situation has occurred.
- Explore Possibilities: Possibly the most obvious benefit, simulation allows difference possibilities to be explored without the need to actually implement them.
- Choose Correctly (Acquisitions): Through exploration, simulation can aid the process of choosing before actually committing to an option.
- Cost Effectiveness: Simulation can help identifying possible problems before a system is implemented. At this point they can be rectified most cost effectively.
- Identify Constraints: As with the previous point, constraints (of a new product design for example) can be identified without actually implementing the system being modelled.
- Safety: Simulators can be used safely as a substitute for actual systems in situations where potentially dangerous actions are involved. Examples include pilots training on flight simulators or military use for exercises.
- Access: Simulations support experimentation for systems that would otherwise be impossible to test (such as those often found in astronomy or cosmology)

From the list above it is clear how simulation can aid in the process of investigation, exploration, training or decision support in many and meaning ways. Given the valuable contribution simulation can make in many situations, its use has become pervasive. Within the wider business community, a number of supporting tools and environments are available, targeting a wide variety of uses and domains.

2.2 Simulation Tools in the Wider Business Community

In pursuit of developing simulation models within mainstream domains, two categories of tools are generally used; generic desktop productivity applications and specialised simulation tools. Each of these tools offers both a number of advantages and disadvantages, which are discussed in this section. Widely recognised as the most popular tool used for simulation purposes within the wider business community [6], spreadsheets are the de-facto standard among desktop productivity applications.

2.2.1 Spreadsheets

The notion of a spreadsheet began in the Accounting domain where they were used to store information regarding the transactions of a business. A spreadsheet was a large sheet of paper organised into columns and rows which was able to "spread, or show" large amounts of related information to a manager for use in the decision making process [94]. Continuing this analogy, a computerised spreadsheet comprises a group of pages, each of which has a table consisting of rows and columns of cells. Each call may contain either data, or a formula (the result of which is presented as the value for the cell) [104]. With a spreadsheet program able to perform automatic calculations based on the contents of a sheet, and their evolution to provide presentation capabilities (such as text formatting and colouring or graphing capabilities) in addition to data processing, spreadsheets have become the most popular desktop productivity tool for small-scale simulation activities [22].

The use of spreadsheet applications as a simulation tool presents many advantages [8, 104]:

- Spreadsheets environments are widely available: models developed by one person are able to be used easily by another
- Spreadsheet environment are able to combine both data processing logic and presentation into a single package
- A large number of built in functions to do mathematical, financial, statistical calculations is provided
- The table based structure allows developers to organise computations and results in an intuitive manner
- Automation of tasks can be achieved through scripting languages and modules developed under traditional programming models
- Spreadsheets provide a generic environment capable of supporting any number of different problems

Spreadsheet applications provide a general-purpose environment that caters well to a larger number of diverse simulation problems. Inbuilt support for common mathematical functions is enhanced by facilities that allow the more complex logic associated with simulation models to be developed via traditional programming languages and plugged in as modules [8].

Combining excellent support for the development of small numerical models with facilities for formatting and presenting the results enables solutions to many small simulation-like problems to be developed in a rapid fashion [104]. However, perhaps the most compelling of all advantages this class of tool presents is their desktop commodity status. General-purpose tools such as spreadsheets are widely available and relatively inexpensive. Their support for a large breadth of purposes has seen them become the de-facto standard for simulation activities in the wider business community [59].

However, while spreadsheet applications provide an environment well suited to the development of smaller solutions, as the simulation models involved begin to grow, the limits of these desktop tools are quickly reached.

Spreadsheet Problems

While the use of spreadsheets as a financial modelling and simulation (M&S) environment does present many advantages, there are also severe drawbacks involved. Primary among these are the problems of structure and development, error, speed and interoperability and reuse.

Structure and Development

While intuitive to use, the structure imposed by the spreadsheet interface creates many problems. Users of spreadsheets enter data into cells, and define formulas for those cells. These formulas reference values contained in other cells for use in calculations, the results of which may be referenced in other formulas contained in other cells [99]. This system of absolute referencing results in a highly in-flexible, interdependent, static structure [59]. Moving even a single value or formula can trigger a cascading effect causing bugs to develop throughout a spreadsheet.

The development of spreadsheet models is often achieved through an ad-hoc, unstructured process [98]. While for small, uncomplicated models this method may be adequate, as larger models are required the effort needed to develop and maintain these spreadsheets becomes significant and expensive. The hard coding of data that occurs from the location centric approach of spreadsheets can also make them difficult and costly to maintain [95].

While spreadsheets provide an intuitive interface and are considered simple to use and master, the actual amount of mental work a developer must go through is considerable [59]. An interface which required a developer to recall or calculate cell coordinates imposes a mental workload which can lead to both the occurrence of serious errors and an increase in development time [29]. Implementing complex algorithms using a spreadsheet environment becomes an unnecessarily complex activity [104], again increasing development and maintenance time. While a major advantage of developing in a spreadsheet environment is the large amount of functions available for a range of purposes, such environments only provide simple data structures for a developer to work with [104].

The interface presented by spreadsheets raises numerous issues relating to flexibility and development; however, comprehension of developed models is perhaps a larger concern. The familiar grid-based interface lacks the expressive power to fully capture and present the often-intricate entities and relationships that exist within a model. Such a situation hinders human comprehension of a model and restricts the environments ability to adequately support developments in which complex entities and relationships must exist. These shortcomings and restrictions complicate the development process and can lead to the occurrence of errors in both the structure of a model and the data it operates on.

Error

The high occurrence rate of errors within spreadsheets is perhaps the single largest problem associated with their use for simulation and decision support mechanisms. While notions of what constitutes an error vary, the general definition is a situation in which an incorrect value is observed or produced [126].

Many studies and field audits have highlighted the high percentage of spreadsheets in which errors were produced. Results of these studies quote figures which identify anywhere from 38% to 77% of surveyed spreadsheets to contain errors of some description [12, 81]. However, of perhaps an even greater concern is that spreadsheets are used in production settings and the results they produced are confidently relied on during the decision making process [15, 99]. The ill-perceived air of simplicity that surrounds spreadsheets can lead users to trust their results, despite an alarmingly high rate of errors.

The high occurrence of errors observed in spreadsheets can be explained by a number of potential problems. The informal development approach generally associated with spreadsheets takes the place of a structured and standardised method. This ad-hoc style involves much less rigor than controlled methods, leading to the introduction of logic errors [80].

Strict dependence on the location of information within a spreadsheet can also trigger the occurrence of errors. Spreadsheets allow users to change the values of cells arbitrarily [31] introducing errors when the other cells depend on certain information being in a specific location. The appearance of information within a spreadsheet provides few visual clues the other information a given cell depends on. This can trigger cascading-style errors when a single value is moved. Further, additional errors can arise when a dependence is created on the wrong cell by mistake.

Finally, as mentioned above, the spreadsheet interface does not posses the expressive power to fully capture complex entities and relationships, in turn hampering human comprehension. This can lead to the occurrence of errors during the development of models involving even a modicum of complexity.

Interoperability and Reuse

Given spreadsheets dependence on the location of information, model composition and reuse also suffer greatly. The ability to create a new entity entirely from pre-existing components is a powerful concept and one that has the potential to reduce the costs involved in composing simulation models. The ability to then reuse these entities in other models generates additional savings and increases the value of such components.

Spreadsheet development is often characterised as a dependence-driven, direct-manipulation process [4] with data often dependent on the location of other data and requires direct manipulation to modify. While many people may use a spreadsheet in its entirety, the reuse of components or sub sections of a model is extremely limited given the dependence on factors like the location of information. A lack of interoperability and reuse leads to the duplication of development efforts where they could otherwise be reused. This results in a higher maintenance costs (as multiple occurrences of similar components must now be maintained) and initial development costs.

Speed

Spreadsheet environments are often slow in their calculation compared to other more specialised modelling environments [104]. Each time a model is executed in a spreadsheet environment all the calculations involved must be translated into a form that can be actually executed. Functions executed in spreadsheets are often interpreted as opposed to pre-compiled; the amount of effort required to have those calculations converted into an executable form can be considerable [104]. This becomes a larger concern when you consider that in certain types simulations, the same model is executed repeatedly while working of differing data. Each time the model is executed the conversion process must again take place. Again, while not an issue for smaller models, when larger and complex ones become involved the situation quickly becomes unattractive.

Moving Beyond Spreadsheets

Due primary to their pervasive nature, simplistic approach and desktop commodity status, spreadsheet applications have become the de-facto standard for simulation activities within the wider business community. While their use offers many potential advantages, there are numerous and significant shortcomings that can hamper development efforts and reduce confidence in the results generated.

Given the problems highlighted above, the achievable size and complexity of developed models is significantly restricted. While there are many advantages to the use of the spreadsheet environment, the size and complexity restrictions mean users forego the in-depth analysis that comes from larger and more comprehensive models.

Moving beyond the limitations and restrictions of the spreadsheet interface, specialised simulation tools can offer several advantages in the pursuit of developing more complex simulations.

2.2.2 Specialised Simulation Tools

While spreadsheet applications are the most pervasive tools used for simulation purposes within the wider business community, their environment raises many potential issues when attempting to develop complex models. However, alternative options do exist in the form of more specialised simulation tools.

Specialised Tool Advantages

Unlike spreadsheets, which are general purpose in nature, specialised tools are designed to address the particular problems of simulation model construction and execution. As such, the development environments they provide are often far more adequately suited to this task. Often graphical in nature (such as [32, 111, 120, 125]) these tools address many of the development shortcomings of the spreadsheet environment. Removing the major drawbacks associated with spreadsheet based simulation development, domain specific applications allow users to work in a setting designed specifically to meet their needs.

Free from the grid-based interface of spreadsheets, specialised tools are able to support simplified methods of model development. The graphical nature of these tools allows greater comprehension and understanding of a simulation model, and provides additional support for novice users lacking an in-depth knowledge of simulation. Such facilities help reduce initial development time and simplify the maintenance process.

Where the complexity of a model is such that the visual environment alone cannot adequately express the desired behaviour, lower-level facilities are provided. As with spreadsheet applications, these tools also allow a developer to utilise programming language constructs to insert additional modules into the model. Often using code generation to create executable models in the background [125], the integration of lower level services is well catered for. When combined, these capabilities result is an environment that aids simulation development from small and simple models to larger and more complex ones.

Specialised Tools Disadvantages

The development environments provided by specialised simulation tools are a significant improvement beyond that of the common spreadsheet. However, despite offering many advantages, this category of applications introduces its own set of shortcomings, hampering their usefulness in many situations.

While specialised, graphical interfaces allow for greater human comprehension of a model a certain level of training is required to learn the customised language of the application. Further, given the proprietary nature of these tools, such knowledge is not portable from one to another. Although such tools provide an environment in which domain experts can be productive and work efficiently, the specialised nature of such tools restricts their usefulness to the situations they were designed for. Spreadsheets however can be useful in much larger array of situations.

Although they provide greater levels of support for simulation model construction and creation, data input has been noted as one particular area where simulation tools suffer [7]. Further, although anecdotal evidence suggests that the use of visual systems can help in the development and validation of simulation models, little published empirical evidence exists to substantiate these claims [7].

Despite the shortcomings identified above, perhaps the biggest obstacle specialised simulation tools present is their proprietary nature.

The large number of commercial modelling and simulation environments competing within the wider business community can cause significant problems when attempting to integrate artefacts developed with alternate tools. Although support may be provided for the reuse of simulation models or model components that were developed within the same tool, any attempt to leverage this investment in another environment is removed.

When attempting to bring together models developed across a variety of domains and specialities, the lack of interoperability between these tools can obscure any potential benefits realised during their development. Ideally, the separate components of large and complex models could be initially developed in the tools that best suited the task at hand. These components could then be assembled with one another, forming a larger co-operative model. However, the lack of interoperability between the myriad of simulation tools available eliminates this option and reduces the reuse potential of the developed models.

Looking over the shortcomings identified above, it becomes evident that the problem of interoperability and reuse is a significant limitation. Whether choosing the pervasive spreadsheet, or the advanced specialised simulation tools, integrating and reusing model components is considerably restricted. Directly affecting the return-on-investment (ROI) made in these models, the lack of reuse and interoperation serves to reduce their achievable size. In turn, the benefits of analysis that models of greater depth can bring are lost; a significant factor when considering that within the wider business community, simulation is used primarily as a decision-support mechanism.

2.3 Addressing the Problems

This chapter has looked at the types of tools used for simulation purposes within the wider business community, in addition to their strengths and weaknesses. The primary problems of interoperability and reuse have been identified as limiting the development of larger and more complex simulations. Addressing these issues is the primary motivation of this research.

Underpinning the development of distributed simulations in the defence community, the High Level Architecture (HLA) has gained acceptance due in part to its support for a broad level of interoperability. The HLA comprises a framework that loosely couples together simulation components developed and deployed on a diverse range of platforms. The HLA has the potential to enable increasing interoperation between otherwise disparate simulations and tools and help address the problems identified above. The HLA, its advantages and disadvantages are discussed in chapter three.

Chapter 3

Distributed Simulation and the High Level Architecture

Where the many types of simulation tools used within the WBC tend to focus on monolithic, single application simulations, distributed simulation partitions the effort into multiple, co-operative units. Underpinning the development of distributed simulations in the defence community, the High Level Architecture (HLA) has gained acceptance due in part to its support for a broad level of interoperability. Encompassing a framework that loosely couples together simulation components developed and deployed on a diverse range of platforms, the HLA has the potential to enable increasing interoperation between otherwise disparate simulations and systems.

Despite enjoying pervasive application within the defence domain, use of the HLA within the wider business community has to this point been minimal. In such domains, where a wide variety of proprietary tools are used for simulation purposes, the HLA can help enable increased interoperability and reuse. Offering a common, standardised, low-level infrastructure, the HLA would allow simulation models otherwise isolated from one another to be used together.

This chapter introduces the HLA, discussing its major components and characteristics. Initially, a brief introduction to the alternate technologies that went before the HLA is presented. Following this, the HLA itself is presented. To conclude the chapter, a brief discussion on how the HLA can help address the problems of simulation within the wider business community is provided.

3.1 Distributed Simulation

Before beginning a discussion of the HLA as the potential solution to the problems identified in chapter 2, it must be established that the HLA is currently that best suited alternative. This section briefly introduces other simulation and application distribution frameworks and discusses their individual advantages and disadvantages.

3.1.1 Common Object Request Broker Architecture

Developed and standardised by the Object Management Group (OMG), the Common Object Request Broker Architecture (CORBA) is an attempt to link together otherwise disparate applications [66]. CORBA allows distributed, heterogeneous applications to communicate with one another in a location and language independent manner. From the perspective of the calling application, all objects Generic Aspect appear to be local. However, the underlying middleware supporting CORBA abstracts the location, language and platform of the remote object, routing requests and responses across application boundaries are required.

The public interface made available by a remote application is described via the Interface Definition Language (IDL). IDL provided a programming-language neutral method for specifying the specifics of an interface and can be used by other frameworks to generate the necessary stub code that will facilitate distributed communication [73].

In addition to IDL, CORBA also defines a generalised communications protocol that allows clients written in any programming language and on any platform to communicate with one another. The Internet Inter-ORB Protocol (IIOP) standardises the format of communications that are to pass between distributed CORBA-enabled applications.

While CORBA supports the distribution of application logic and is capable of enabling greater levels of interoperability between otherwise disconnected applications, it is a general solution and does not provide support for common simulation functionality. Advanced simulation services such as integrated time management, interest specification (publication and subscription), ownership management and data distribution services are all unavailable. However, it is important to note that some existing HLA implementations have used CORBA as a communications protocol. Although HLA infrastructure tools like RTI-NG and GERTICO are based on CORBA, it is up to these particular implementations to provide the advance simulation services themselves. For this reason, CORBA alone is not a suitable distributed simulation platform.

3.1.2 Remote Method Invocation

Remote Method Invocation (RMI) shares many similarities with CORBA. Its primary purpose is to allow the invocation of methods on distributed objects (ones that do not exist in the same memory space, or even the same computer, as the executing program). Developed by Sun Microsystems, RMI was initially intended to be a solution that only supported the Java programming language. However, more recent versions have seen support added for the IIOP protocol utilised by CORBA.

As with all application distribution technologies, facilities to specify the methods and parameters that are callable in a distributed manner are provided (via Java interfaces) in addition to services for locating and connecting to the distributed providers. However, as with CORBA, RMI was designed to be allow the invocation of remote functionality in a location transparent manner. As such, it lacks specific support for simulation activities (such as co-ordinated time and message delivery).

While both CORBA and RMI provide robust support for the interoperation of software applications in a distributed environment, they lack support for common facilities found in simulation-specific frameworks. DIS and ALSP (described below) are two examples of distributed simulation frameworks that go beyond generalised application distribution.

3.1.3 Distributed Interactive Simulation

The Distributed Interactive Simulation (DIS) framework is an IEEE standard (IEEE 1278) that began primarily as a means of connecting various large, human-in-the-loop simulators (eg. flight simulators). The DIS framework centres on a standard set of Protocol Data Unit's (PDU) that describe the format of messages that can be exchanged between participating components. When certain state changes within a given simulator occur (such as the movement of an entity), these messages are broadcast to all other participants. The use of dead-reckoning algorithms helps to reduce the amount of network traffic by allowing simulators to send less frequent updates while remote clients interpolate information about the position of an entity based on their previously provided information (such as speed and heading).

While DIS has proven successful in linking together and allowing the interoperation of many disparate platforms, certain problems render it unable to help address the problems highlighted in chapter 2. DIS lacks any notion of centralised time co-ordination or the ordering of events. As such, simulation repeatability is not possible. This is not a problem in the virtual worlds DIS was designed to support. However, this is not suitable for analytical-style simulations such as those used commonly in the wider business community for decision support.

DIS primary is a protocol designed to standardise communications between various military simulators. PDU's only exist for concepts that make sense in a physical world (such as the movement of an entity). Attempting to link together financial simulations would require the specification PDU's describing the salient state changes that might occur in that context. DIS is very tightly defined, and as such, it is unable to provide any support in a generic context.

3.1.4 Aggregate Level Simulation Protocol

The Aggregate Level Simulation Protocol (ALSP) is one of the closest predecessors to the HLA. Similar to DIS, ALSP describes a protocol for messages that are to be passed between the various participants of a distributed simulation [123]. Moving beyond DIS, ALSP provides global time synchronization [67], helping to address the causality and repeatability issues of DIS. Much like HLA, the shared object model of a distributed simulation takes on an object-oriented approach, modelling information as objects with attributes.

Although filtering is provided, advanced interest management facilities are not provided and all information changes are still broadcast to all participants. Containing a number of similarities with the HLA, ALSP is perhaps best regarded as a subset. While supporting many of the same features, certain highlights are still missing (e.g., time management among different kinds of simulations and data distribution management) [112]. While well aligned with the problems raised in chapter 2, the level of functionality provided by ALSP is a subset of that provided by the HLA. Further, unlike the HLA, ALSP has no open international standard and has not enjoyed the same substantial ongoing research and development that continues to surround the HLA.

3.1.5 Summary

The various technologies introduced above are each aimed at joining together separate applications or simulation in a distributed fashion. CORBA provides for location-independent interoperability among heterogeneous applications, while DIS and ALSP have proven capable of bringing together disparate simulations. However, in recent years the HLA has emerged as the most prevalent distributed simulation framework. The following section introduces the HLA; its processes and components.

3.2 The High Level Architecture

The HLA was initially developed by the United States Defense Modeling and Simulation Office (DMSO) in order to address the need for interoperability between simulations (both new and legacy) used within the US Department of Defense (DoD). The HLA aimed to extend upon the work surrounding the Distributed Interactive Simulation (DIS) and the

Aggregate Level Simulation Protocol (ALSP) [37] and provide a standard framework for simulations used within the DoD.

In 1996, the HLA was mandated for use in all new works purchased by the DoD, however, recent times have seen this requirement loosened somewhat [118]. In the interests of developing a vibrant and active community, the HLA was adopted as an IEEE specification (IEEE 1516) in 2000 [47, 48, 49, 61], thus providing an open process for contributors to become involved in the standardisation effort. Despite significant work from a large number of people, the original IEEE 1516 standard contained numerous ambiguities and shortcomings. This in turn led to the development of companion specifications [42, 106, 107] that addressed the problems by extending the standards. Currently, the IEEE specification is under periodical review with enhancements and extensions being made. Due for ratification in late 2006, the new "HLA-Evolved" standard will address the problems previously identified and provide an improved, unified, open specification.

3.2.1 HLA Overview

Within the HLA, individual simulation components, known as *federates*, exchange data and work together in a *federation*. Communication and co-ordination between the separate federates is handled via a central component known as the Run-Time Infrastructure (RTI). Figure 3-1 provides a logical overview of this structure:

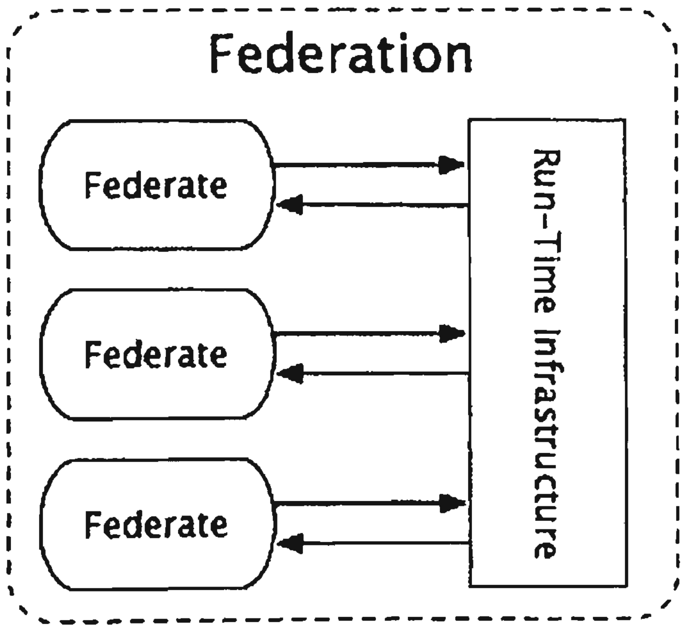


Figure 3-1: Federation Overview

Conversations between a given federate and the RTI are bi-directional. Federates communicate with the RTI through a standardised interface known as the

RTIambassador. This interface comprises the set of services that provide specialised simulation functionality (time management, publication and subscription, data exchange facilities, etc...). When the RTI needs to pass information to a given federate, it can do so via another standardised interface known as the FederateAmbassador. The implementation of this interface is provided by the federate, thus enabling it to take action on any incoming information. Figure 3-2 shows the components involved in these communications:

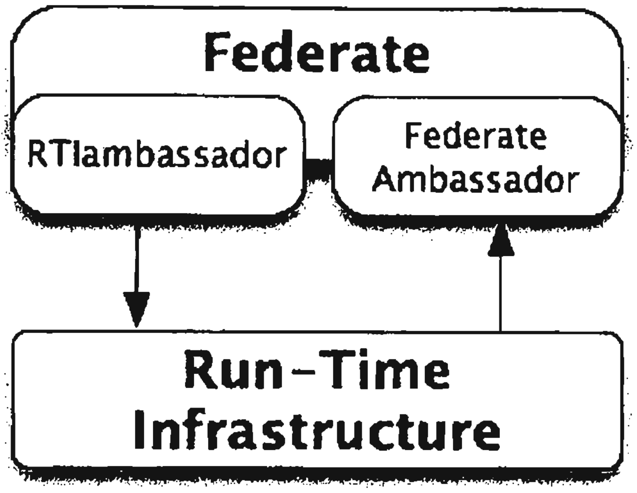


Figure 3-2: Federate Communication

When a given federate wishes to provide updated information to other parties involved in the federation, it does to by informing the RTI ambassador of the new values. This information is then filtered such that only the appropriate portions are passed to a given federate, where they receive notification of the update through the their federate ambassador.

All information exchanged between federates must conform to a common object model. The Federation Object Model (FOM) establishes the shared vocabulary of a federation. Each federate within a federation is itself considered to be a smaller simulation. As such, an individual document, known as the Simulation Object Model (SOM), defines the structure of information each federate produces and consumes.

The following sections provide a more detailed look at both object models and the standard interfaces/services used within the HLA.

3.2.2 HLA Object Models

The primary purpose of object models within the HLA is to define and document the structure of information that is of interest to either a specific federate or federation. The transmission of all information between federates occurs as an opaque series of bytes [42]. Given this, object model information is a vital component of any HLA simulation, providing an instruction manual that enables shared information to be reconstructed into some meaningful form. Depending on the model in use, additional information such as the intentions of a given federate to produce or consume certain information defined by the model is also provided [114]. The Object Model Template (OMT) defines the format these models must conform to [49].

3.2.2.1 Simulation Object Models

In the HLA, each federate has its own object model known as its SOM. Each federate within a distributed simulation is itself considered an individual smaller simulation, thus, the SOM for each federate describes the object model it uses. Beyond describing the structure of information for the federate, a SOM may also describe which of the entities it intends to provide to other federates and which it desires to consume information about [49].

While the SOM defines the object model for a particular federate, internally the federate does not have to work with information structured the same way. The SOM simply describes the public face of the federate [112]. Although the HLA specification defines that each federate must have a SOM [50], during execution the document is generally never used. Despite this, the SOM is still a vital documentation component and is often used by middleware or code-generation frameworks that seek to provide simplified methods for developing HLA federates (such approaches are discussed later in this document).

3.2.2.2 Federation Object Models

While a SOM describes the object model for a given federate, a FOM describes the shared vocabulary for a federation. Broadly speaking, a FOM can be thought of as an intersection of the SOMs for all the participating federates. This is not strictly true as any federate can choose to produce or consume only portions of the information defined in a FOM[1]. Regardless, a FOM formally defines the structure of all information that is available to be **passed between federates** and communication regarding information not contained within it is prohibited [50].

Given that a FOM defines a shared lexicon, it is one of the primary vehicles enabling simulation interoperability within the HLA [112]. Any federate that is able to produce information according to a particular FOM is able to communicate with any other such federate via the RTI. Given this, the compatibility between a given SOM (describing the

[1] Further, a FOM may have elements that are neither produced nor consumed by any federate.

object model of a federate) and a given FOM is a crucial aspect when attempting to enable broad-level interoperability. While the OMT format provides a shared grammar for these documents, bridging semantic gaps is far more difficult. The HLA provides the syntax for interoperability [112], solving semantic differences is left up to the user.

Object models within the HLA draw heavily from typical Object-Oriented (OO) approaches. While the two are not an exact match, significant areas of overlap exist. As with OO approach, one of the primary constructs of a HLA object model is the object class.

3.2.2.3 Object Classes and Attributes

As with traditional OO approaches, information within a HLA object model is organised in an *Object Class* hierarchy. Object classes describe the entities simulated by federates, and may contain any number of *attributes* (including zero). Attributes are the primary mechanism for specifying persistent storage information. The hierarchies described within the object models define the parent-child relationships that exist between classes. This is important as the OO concept of *inheritance* applies to the attributes of an object class. For example, consider Figure 3-3:

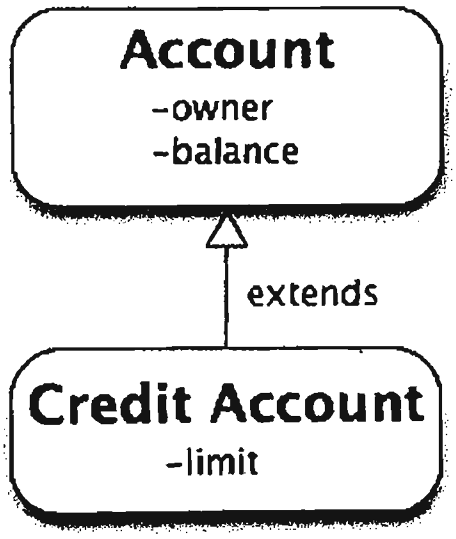


Figure 3-3: Attribute Inheritance

In this example, the `CreditAccount` class would contain three attributes: the explicitly defined `limit` attribute, and the `owner` and `balance` attributes inherited from `Account`. Instances of the object classes defined in an object model are the primary modelling entity within the HLA. As object instances are created, the values of their various attributes can be set and altered.

While traditional OO concepts apply to object class hierarchies and attribute inheritance, the HLA has no direct equivalent for method specification [62]. Although the HLA supports the ability to pass messages between federates via Interactions (discussed next), these are not directly associated with either a specific class of object or an individual instance.

3.2.2.4 Interaction Classes

As mentioned above, while the HLA has no specific notion of methods in the traditional OO sense, it does provide facilities that allow the passing of transient messages [26]. The central entity involved in this process is the *Interaction Class*. As with object classes, a hierarchy of interactions if defined within the object model. Each interaction class may prescribe a number of parameters that contain the values of a message. As with attributes, the principle of inheritance applies to parameters.

While their primary purpose is the passing of messages between federates (perhaps to trigger additional processing or signal an event), interaction classes share many differences with OO-style methods. Interactions are not associated with a given object class or instance. Further, the HLA provide no support for directly targeting a specific federate with an interaction, rather, any federate that signals an interest in an interaction may receive them.

As any federate can show an interest in a specific class of interaction and as such, the direct passing of messages from one federate to another is not explicitly supported. However, if this behaviour is required, federates can be programmed to support it (perhaps through the inclusion of a parameter identifying the target federate, which in turn causes other federates to ignore the message). Just as the direct passing of messages from federate to federate is not supported directly (as is typical with OO-style methods), workarounds that include an identifier for the target object instance can be programmed into federates should the developers desire it.

3.2.2.5 Data Types

Following the IEEE 1516 standardisation process, support was added to the OMT format for defining data types [49]. When information is passed between federates (via attribute updates or interactions for example), the values of the attributes and parameters concerned are formatted as an opaque series of bytes. The IEEE 1516 specification provides support for associating a given data type with an attribute or parameter, thus fully defining their structure.

Building on primitive types, the OMT specification provides support for more complex arrangements in the following formats [49]:

- **Simple Types:** Generally speaking, simple types are just restrictions or associations with a given primitive type. That is to say, the simple type `Minute`, might define an integer that can be of the value 0 to 59.
- **Enumerated Types:** Used to define a "data element that can take a finite discrete set of values" [49].
- **Array Types:** Used to define collections of another data type. These may be static in size or dynamic.
- **Fixed Record Types:** Define a complex type that may consist of many other types. For example, the type `Position` may be defined as containing three consecutive 64-bit floating-point values defining an x, y and z value.
- **Variant Record Types:** Variant records describe "discriminated unions of types" [49].

3.2.3 The HLA Interface and Processes

When exchanging information about the creation, alteration or removal of data described within the FOM, each federate communicates with the RTI via a standard interface. This interface defines how each federate can access the various simulation services provided by the RTI, such as time, declaration and object management. This section introduces the mechanisms involved in exploiting these services.

3.2.3.1 HLA Interface Facilities

The HLA interface exposes many facilities, each of which can be grouped as follows (the relevant services are discussed in more detail later in this section):

- **Federation Management:** These services cover the management of a specific execution, enabling the creation and removal of federations, in addition to allowing federates to join to and resign from a federation. Additionally, synchronization facilities allow federates to co-ordinate their execution at certain named points, while save and restore functionality allows federates to return a simulation to a previously defined state.
- **Declaration Management:** These services allow a federate to inform the RTI of its intentions to produce and consume the various entities defined in the FOM.
- **Object Management:** The Object Management facilities allow a federate to register, update and remove instances of the various object classes defined within a FOM. These services also cover the ability to send and receive interactions.

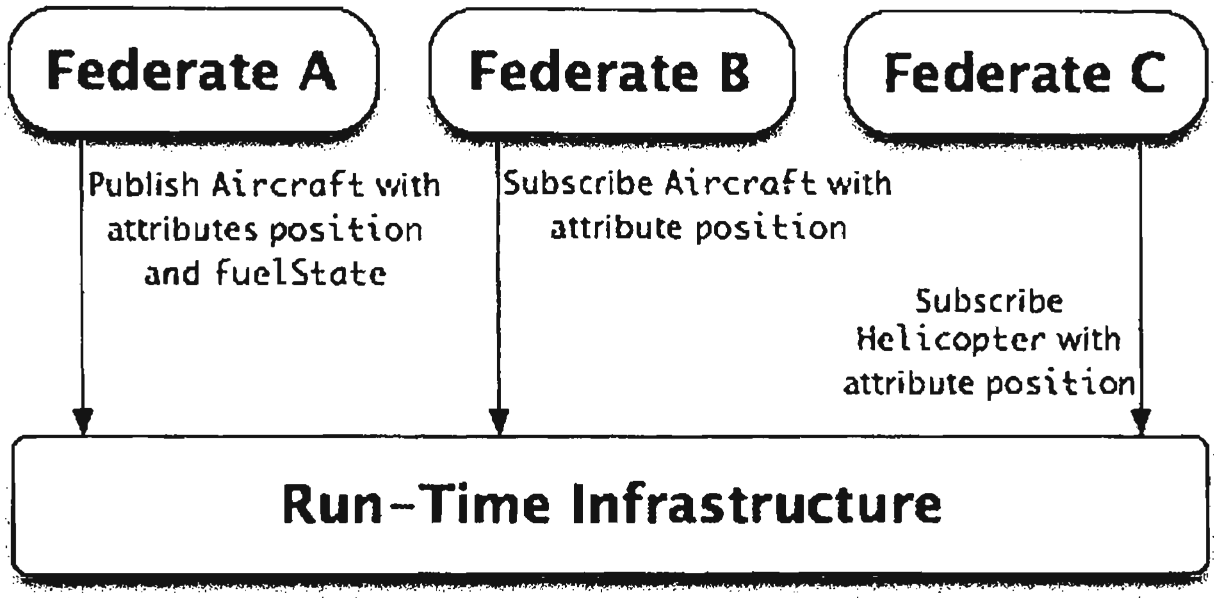
- **Time Management:** The Time Management services allow a federate to control and advance logical time within a simulation. Support is provided for defining whether or not a federate will produce events which are time stamped, and whether a given federation intends to be constrained by the current time status of other federates.
- **Ownership Management:** When information is created by a given federate, it is implicitly granted ownership over it. To alter attribute values, a federate must be the owner of them. The Ownership Management services provide facilities to obtain and transfer the ownership of attributes between federates.
- **Data Distribution Management (DDM):** Providing a mechanism to reduce both the transmission and reception of irrelevant data [72,112], the data distribution services allow a federate to specify a region in which a subscribed attribute must be in order for the federate to receive notification of changes. When a federate updates an attribute, it can outline the given region in which that update is relevant. Unless the update region intersects with the subscription region for a different federate, it will not receive that update. DDM provides more fine-grained control over attribute updates (or interactions), reducing network traffic to a minimum.

The HLA standards define realisations for these services in three different programming languages: Java, C++ and Ada [48]. While these are the only standardized mappings, there is nothing preventing federates written in other languages or software platforms from forming part of a distributed simulation. Having introduced the various sections of the standard HLA interface, some discussion of the relevant salient details is required.

3.2.3.2 Publication and Subscription

Publication and subscription are the means by which federates signal to the RTI their intent to produce and consume state information of particular types declared in the FOM. While older simulation frameworks such as DIS and ALSP used a broadcast mechanism to distribute state changes [123], the HLA allows a federate to show selective interest without needing to implement filtering itself.

Consider Figure 3-4 shown below:



Federate A
Federate B
Federate C
Publish Aircraft with attributes position and fuelState
Subscribe Aircraft with attribute position
Subscribe Helicopter with attribute position
Run-Time Infrastructure

Figure 3-4: Publish and Subscribe

This figure shows three federates, each with differing publication and subscription interests. Federate A signals to the RTI that is intends to publish the `position` and `fuelState` attributes of the `Aircraft` class. Federate B informs the RTI that it only wishes to hear about state changes to the `position` attribute of the `Aircraft` class and Federate C only to the `position` attribute of the `Helicopter` class.

Until the point at which Federate A declares that it wishes to publish the given attributes of the `Aircraft` class, any attempt to create a new instance of this type will be forbidden by the RTI. Accordingly, unless Federate B signals that it is interested in the `position` attribute of the `Aircraft` class, it will never receive updates about changes made in other federates. Publication and subscription services are dynamic. Thus, during any point within a simulation, a given federate can decide to publish, unpublish, subscribe or unsubscribe various pieces of the FOM.

These facilities allow the RTI to filter incoming information and pass only the relevant portion to interested federates, rather than broadcasting all messages and burdening each individual federate with the task of filtering it [72,67]. The effect publish and subscribe calls is demonstrated in the next sub-section.

With one small exception, the semantics of publication and subscription apply to interactions just as they do to object classes and attributes. Where a federate can signal publication or subscription interests for specific attributes of an object class, with regard to interactions, the action applies to the class as a whole (not individual parameters).

3.2.3.3 Information Creation and Distribution

Once a federate is either publishing or subscribing to various attributes or interactions, it will start receiving information about the relevant events that are occurring in other federates. The list below shows the four different types of events recognised by the HLA:

- **Object Instance Registration:** Creating an specific instance of an object class contained in the FOM
- **Attribute Value Alteration:** Changing the value of a particular attribute contained within a particular object instance
- **Object Instance Removal:** Deleting a specific object instance
- **Interaction Sending:** The transmission of an interaction

Object Instance Registration

All persistent simulation data within a federation is stored in attributes contained within object instances. Before a federate can register an instance of an object class, it must be publishing at least one attribute (either declared or inherited). Continuing the "three federate" example from Figure 3-4, Figure 3-5 demonstrates what happens when an object instance is registered.

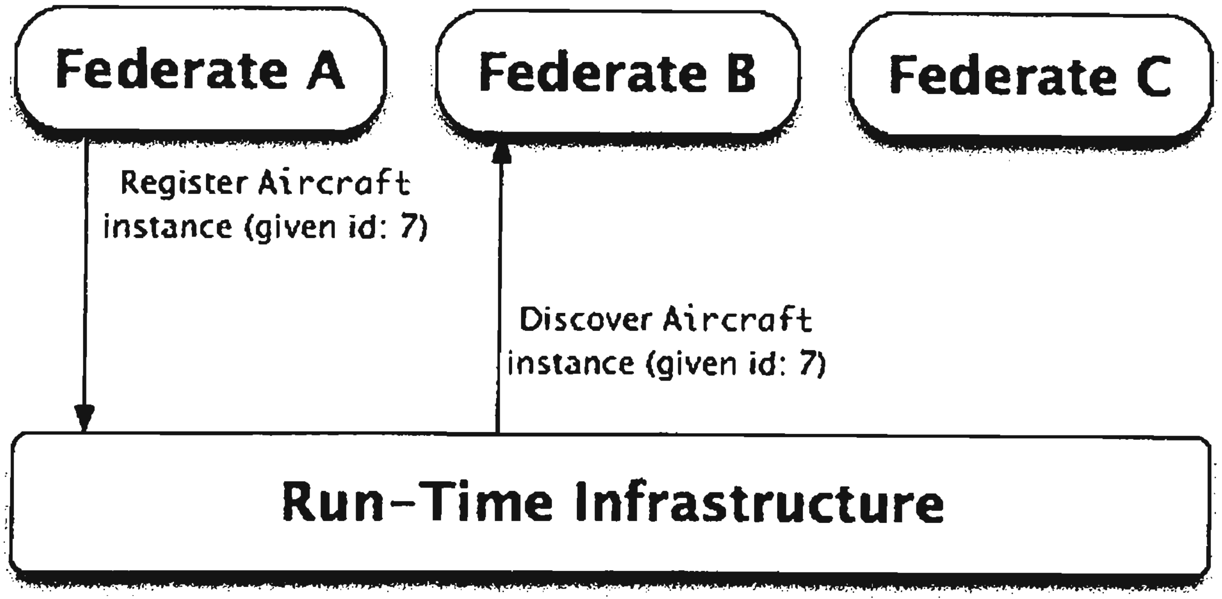

Federate A
Federate B
Federate C
Register Aircraft instance (given id: 7)
Discover Aircraft instance (given id: 7)
Run-Time Infrastructure

Figure 3-5: Object Instance Registration

Here, Federate A informs the RTI that it is registering an instance of the Aircraft class. As Federate B is subscribed to at least one attribute of this class, the RTI notifies it that an instance has been created via a *discover* call back. Given that Federate C has no interest in the Aircraft class, it will not receive any information about this new instance.

Type Promotion

Type promotion refers to the situation in which an instance can be discovered (or an interaction received) as a different type to that which it was sent as. For example, if Figure 3-5 contained a fourth federate that subscribed to a `Vehicle` object class (where `Vehicle` is the parent of both `Airplane` and `Helicopter`), that federate would discover any instances of either child class that were registered. However, rather than discovering the instance as the type it was registered as, the fourth federate would see any instances as types of `Vehicle`. In this case, the child classes had been *prompted* up the hierarchy [27]. Further, the federate would only be able to see any attributes of the `Vehicle` class (and **none** of the child classes).

The same is true for interactions. If a federate is subscribed to a given interaction class, and an interaction of a child-class is sent, the federate will receive it as the parent class (with only the parameters that are relevant for that class).

Attribute Value Alteration

Over the course of a simulation it is expected that the various values of certain information will be updated. During the simulation run, a federate can alter the value of the attributes associated with a previously registered object instance. Figure 3-6 demonstrates this process:

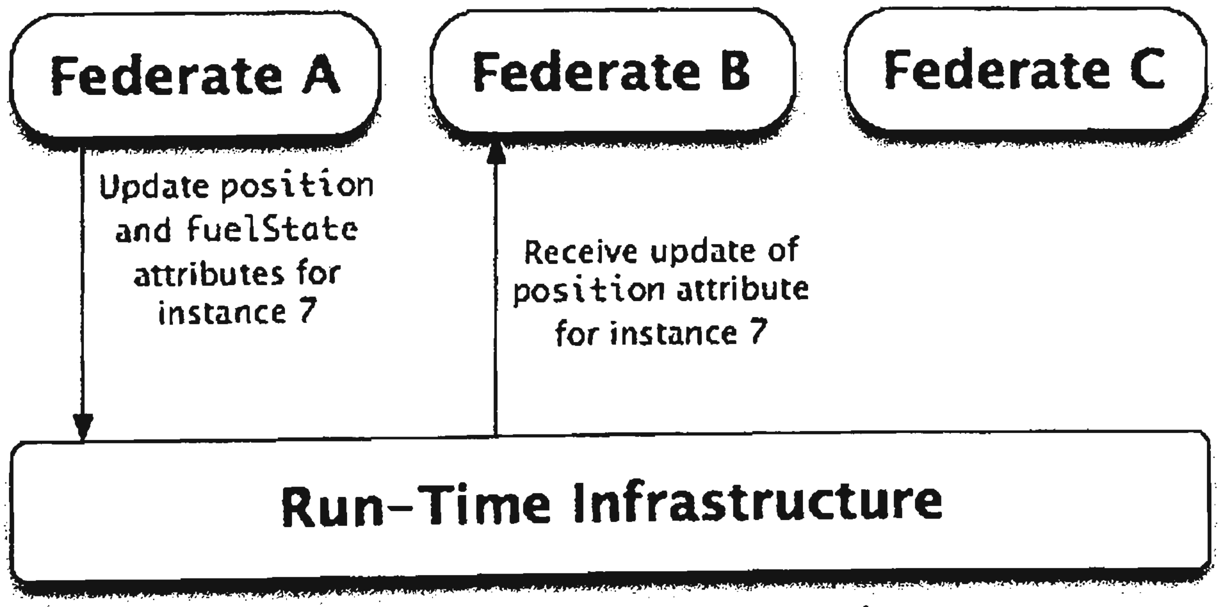

Federate A
Federate B
Federate C
Update position and fuelState attributes for instance 7
Receive update of position attribute for instance 7
Run-Time Infrastructure

Figure 3-6: Attribute Value Update

Having changed the value of the position and fuelState attributes, Federate A notifies the RTI. As Federate B is only subscribed to the position attribute, it will receive information only of its alteration. Again, as Federate C is not interested in any relevant information, none shall be passed to it.

Object Instance Removal

At some point during a simulation run, previously created object instances may need to be removed, and thus, all federates using that information notified. The process of removing an object instance is much like that of creating one or updating attributes:

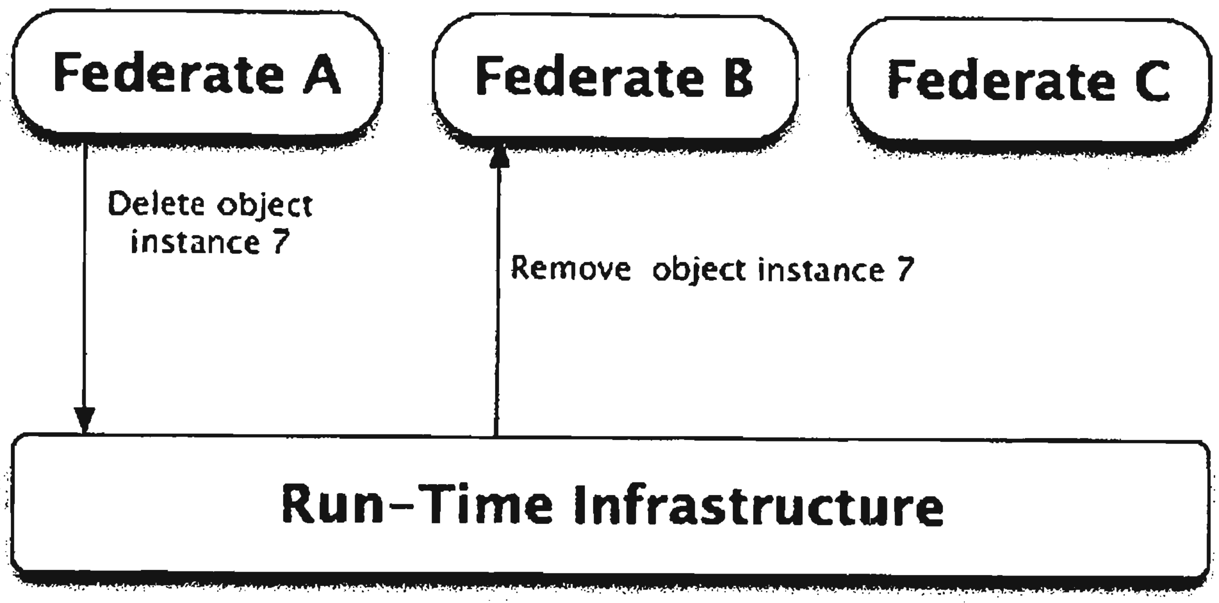


Figure 3-7: Object Instance Removal

Figure 3-7 shows Federate A informing the RTI that the previously created Aircraft object instance should be removed. As Federate B has previously discovered this instance, it is informed of the removal. Given the subscription interests of Federate C, it never discovered the instance, and as such, it does not receive any notification if the removal.

Interaction Sending and Receiving

The final mechanism for information distribution within the HLA is interaction sending. Designed to model transient messages, interactions (unlike object instance attributes) are not meant to represent persistent data. The typical notions of publication and subscription apply to interactions, with the exception that interest is shown on a class level (rather than the lower parameter level) [27]. It is also important to note that when sending an interaction, a federate does not need to provide values for all the parameters [112].

Figure 3-8 shows the publication and subscription interests of three federates, in addition to the hierarchy of interactions in use:

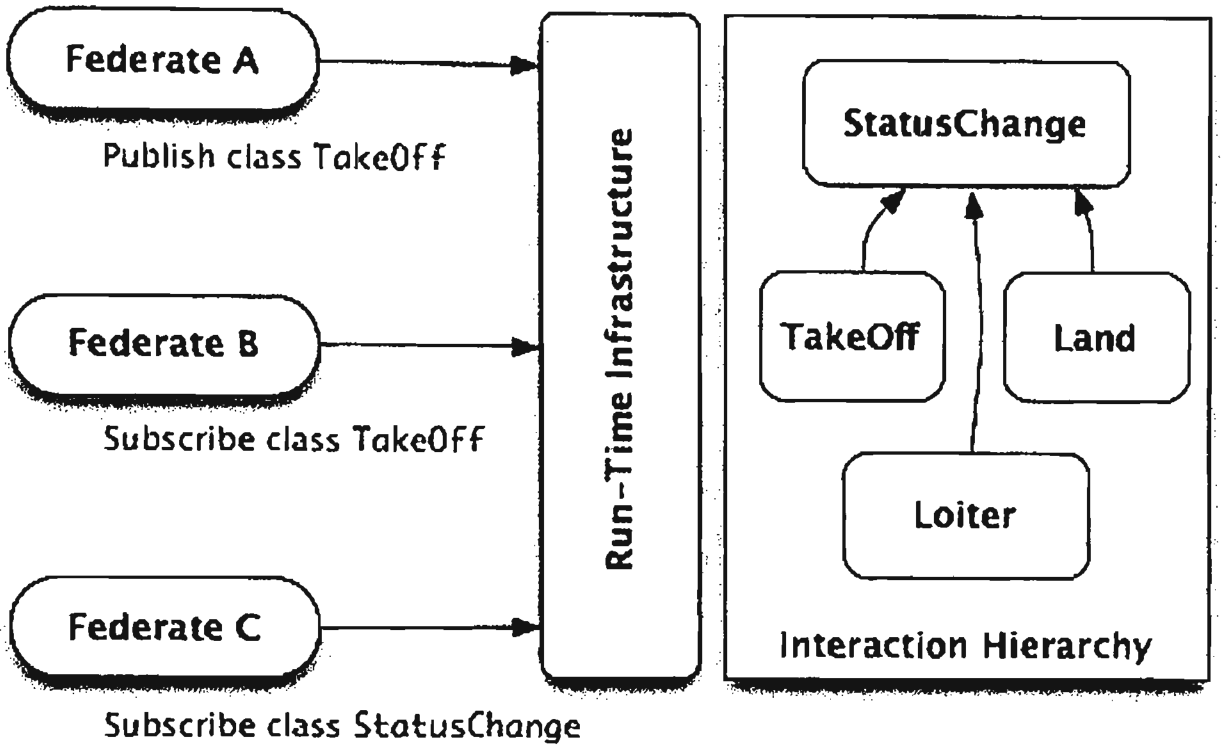


Figure 3-8: Interaction Publish and Subscribe

Here, Federate A is publishing the interaction class TakeOff and Federate B is subscribed to it. In this case, Federate C has subscribed to the parent class, StatusChange. Figure 3-9 demonstrates what happens when Federate A sends an interaction:

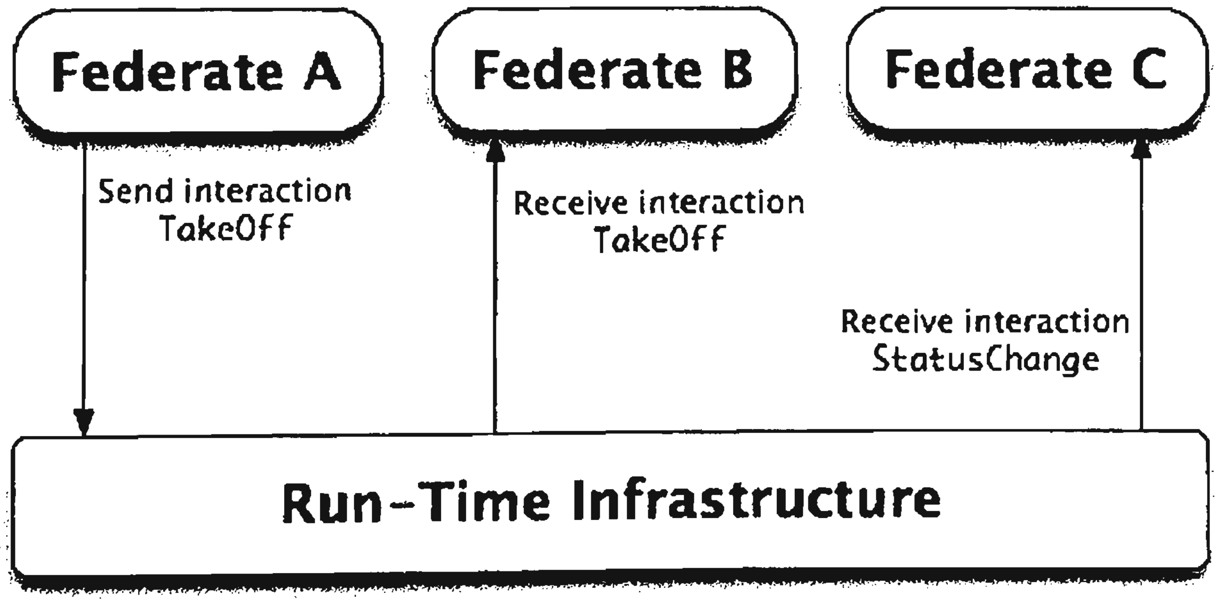


Figure 3-9: Send Interaction

Federate B is subscribed to TakeOff, thus, it receives the interaction as is. However, as Federate C is subscribed to StatusChange, type promotion means that it will receive the interaction as that type (and without any parameters that were introduced by the TakeOff interaction class in the FOM).

3.2.3.4 Time Management in the HLA

The concept of time is central to a vast number of distributed simulations. One of the primary advantages of the HLA over previous distributed simulation frameworks is the shared time management facility it provides [14]. Throughout a given simulation run, these services allow separate federates to remain synchronized to a central "logical time" and help guarantee the delivery of certain messages at a specific logical time.

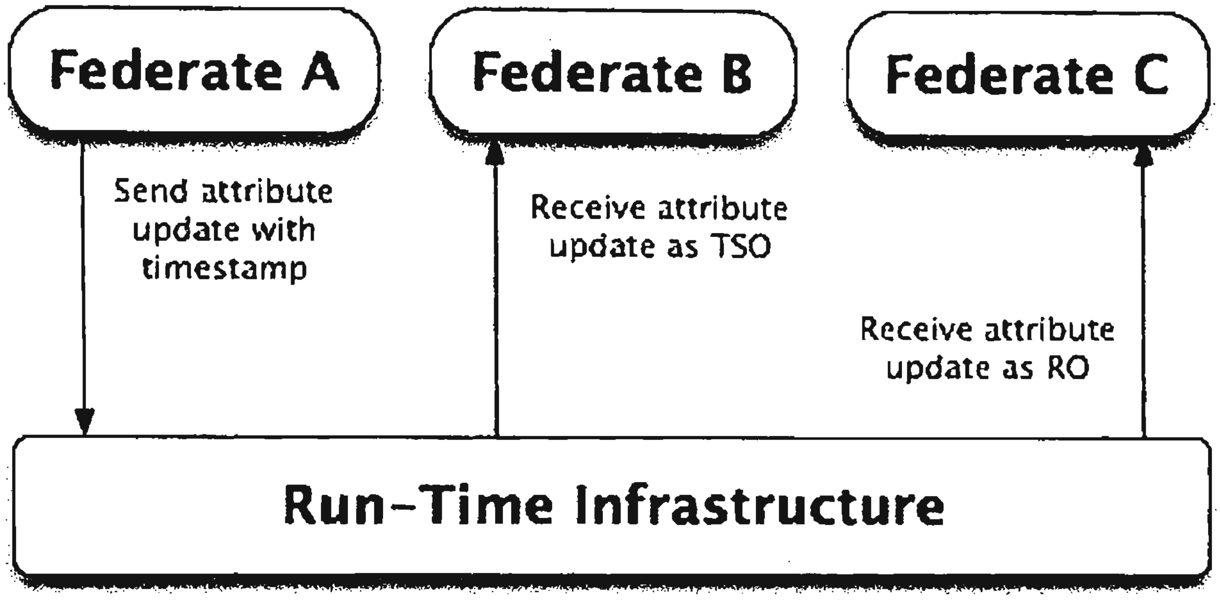
Logical, or *Simulation* time is the logical measure of time within a simulation. Logical time is a synthetic measure whose values are somewhat arbitrary. One unit of logical time might represent one second or one hour of actual time depending on the simulation. The time services provided by the HLA focus on time advancement and the association of messages with a specific logical time.

Logical Time and Message Delivery

Messages within the HLA are delivered to a federate in one of two mechanisms: Receive Order (RO) or Timestamp Order (TSO). RO messages are simply queued up and delivered to a federate as soon as possible, while TSO messages are only released to a federate once the RTI can guarantee that no messages with an earlier (lower) timestamp will be created [112]. The mechanisms used to determine when this point has been reached are introduced during the time advancement discussion below.

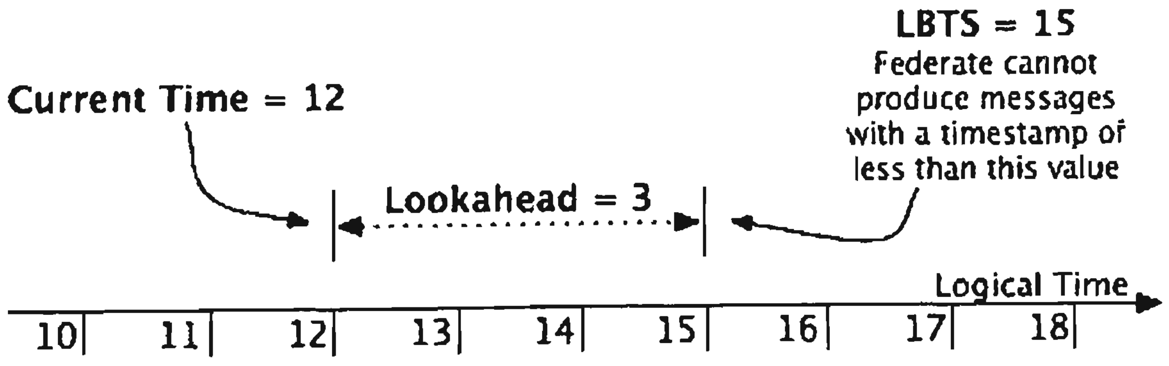
In order to help the RTI manage the delivery of time stamped messages, federates can signal that they wish to be either time *regulating* or time *constrained* (or potentially both or neither). Time regulating federates are those that wish to produce messages (such as attribute updates) with an associated timestamp [48]. Time constrained federates are signalling to the RTI that they wish to receive messages in timestamp order [48].

Only time regulating federates can send TSO messages and only time constrained federates can receive them [27]. If a non-regulating federate attempts to send a TSO message, the timestamp will be discarded and the message delivered as RO [48]. Accordingly, if a non-constrained federate receives a message that was sent with a timestamp, it will be discarded and the message delivered as RO (to that particular federate) [48]. Figure 3-10 demonstrated this process:


Federate A
Federate B
Federate C
Send attribute update with timestamp
Receive attribute update as TSO
Receive attribute update as RO
Run-Time Infrastructure
Figure 3-10: Time Constrained and Regulating

It is important to note that the non-constrained federate will receive the attribute update as soon as possible, while the constrained federate will have to wait until the appropriate logical time has been reached (which may not occur for quite some time) [14].

To help the RTI ensure that no TSO messages are delivered in the logical "past" (with timestamps lower than the current logical time), regulating federates must provide a "look ahead" value. The look ahead is added to the current logical time to determine the lowest timestamp a federate may associate with a message. This value is known as the Lower Bound Time Stamp (LBTS) [14]. Figure 3-11 demonstrates how look ahead affects the ability of a federate to send a TSO message:


LBTS = 15
Federate cannot produce messages with a timestamp of less than this value
Current Time = 12
Lookahead = 3
Logical Time
10
11
12
13
14
15
16
17
18
Figure 3-11: Lookahead

The RTI will not release any TSO message to a constrained federate until it can determine that no more messages with a timestamp less than the proposed new logical time will be generated [27]. To enable the RTI to determine this point, federates use the various time advancements services provided.

Time Advancement

To enable the proper coordination across a federation, the RTI controls when time constrained federates are allowed to advance their logical time. A given federate must explicitly request an advancement in its logical time and wait for the RTI to grant its request before moving on [112]. Consider figure 3-12 below:

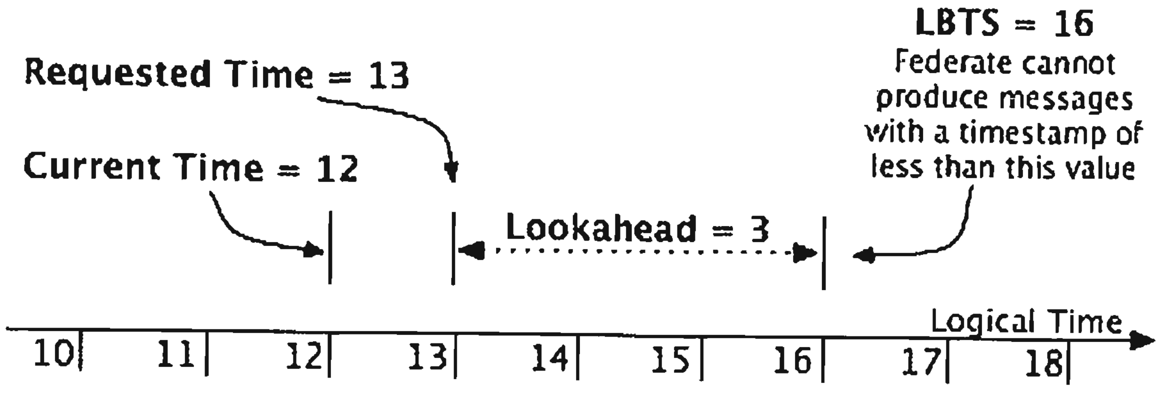


Figure 3-12: Time Advance Request

In this figure, the current logical time for the given federate is 12, however, it has requested a time advance to 13. A request to advance time is taken as an indication that the federate no longer wishes to produce messages at the current time. Thus, if the federate is regulating, its look ahead will now be calculated from the requested time (making the LBTS for this federate equal to 16) [27].

The RTI will not grant this advance until it can determine that all TSO messages with a timestamp of less than 13 have been delivered. To do this, it must determine the earliest possible timestamp a message might arrive with; it needs to know the LBTS for the *federation*. As regulating federates are the only ones capable of producing TSO messages, this value will be equal to the lowest LBTS of all such federates. Only when this value is greater than or equal to the requested time will the RTI grant the advancement[2].

The HLA provides three different methods for the advancement of logical time:

[2] While the HLA provides facilities to coordinate time advancement, an independent logical time is maintained for each federate. For example, it is valid for one federate to have a logical time of 12 while another has a logical time of 15. The RTI will only restrict the advancement of constrained federates as they are the only ones capable of receiving TSO messages. Advancement requests by non-constrained federates will be granted immediately.

- **Time Stepping:** Federates directly request advances of a given value. Generally speaking, the value is consistent (i.e., Always requests advances of 3 units)
- **Event Based:** Rather than directly managing the time advancement process, these federates are more interested in processing the next available message (whatever timestamp it might have). As all TSO messages will be delivered in order, requesting the next event is implicitly asking the RTI for an advance to the timestamp of that message [48].
- **Optimistic:** Time-stepped and event-based approaches are known as conservative techniques [112]. This is because they guarantee that events will be delivered in timestamp order and that no events will occur in the past. Optimistic federates are able to receive all currently queued-events, even if there is the possibility that more messages with smaller timestamps may still be generated [48]. Facilities are also provided to allow such federates to cancel previously sent messages if required.

A major strength of the HLA is that it allows the combination of federates using different advancement mechanisms. From the perspective of a federate, the mechanisms used by other federates are hidden and irrelevant, allowing, for example, time-stepped federates to interoperate with event-based and optimistic ones [112].

3.2.3.5 Ownership Services

Each attribute of a given object instance is explicitly owned by a single federate. Only the federate that owns an attribute may update it. The ownership management services provided by the HLA allow for the transferral and acquisition of attribute ownership among federates. These facilities support the co-operative modelling of simulation data by allowing separate federates to manage various attributes of the same instance, or, indeed, share management of a specific attribute [27].

The HLA supports two types of attribute transferral: push and pull. A *negotiated* attribute ownership transferral is used when a particular federate wishes to "get rid" of (or push away) a given attribute [112]. A *requested acquisition* is used when a given federate wishes to gain ownership (pull) of an attribute currently managed by another federate.

3.2.3.6 Data Distribution Management

With the HLA, publication and subscription mechanisms are used to reduce the amount of unnecessary traffic flowing between the RTI and federates. Data Distribution Management

(DDM) provides facilities that allow more fine-grained control over updates and further reduce superfluous communications.

When a federate is subscribing to a set of attributes or an interaction class, it may also supply a specific region. Regions are multi-dimensional spaces that define constraints for the transmission of events. When a federate updates an attribute value, it may supply an *update region* with the event that signals the space it is valid for. Only when that space overlaps with the subscription region for given federate that is interested in that attribute, will the RTI forward the update. This removes the transferral of information that the receiving has declared itself as having no interest in [72,27].

3.2.4 Summary

This section introduced the HLA, its component, entities and processes. Discussion has focused on both object models and the various simulation services provided by the HLA that enable the interoperation of federates within a distributed simulation. With this in mind, the section 3.3 discusses how the interoperability benefits of the HLA can help remedy the problems of the wider business community as highlighted in chapter 2.

3.3 HLA for the Wider Business Community

The landscape of tools used for simulation purposes with the wider business community consists of numerous and varied options. From general-purpose spreadsheet applications to specialised simulation environments, the lack of interoperability between the produced models can have considerable effects on reusability and return on investment. Designed specifically to address the problems of connecting many varied hardware and software platforms together, the interoperability afforded by the HLA presents significant attractive options when considering it for use in such a diverse landscape. This section describes how the HLA can be of benefit.

3.3.1 Size and Complexity

Simulation tools used within the wider business community tend to be monolithic in nature. The use of a single tool producing a single simulation can negatively impact both the development complexity and achievable size of a model. Distributed simulation focuses on the development and co-operation of multiple, smaller and more specialised models. Partitioning the required work into smaller, more manageable units helps reduce development complexity and allow for greater reuse through the integration of pre-written components.

Such an approach also allows for the integration of additional modules at later points, increasing the overall size of the model and potentially enhancing the information is produces. As the single simulation, monolithic approach places restrictions on the achievable size of a model. The increased depth of analysis extending from lager, more compensative models is forgone in an environment like that which currently exists within the wider business community. Distributed simulation offers enhanced scalability, thus addressing these shortcomings.

3.3.2 Interoperability and Reuse

As mentioned above, the wide variety of tools used for simulation within the wider business community can negatively affect the return on investment made in developed models through reduced reuse and higher initial development costs when developing complex models. Further, the lack of reuse caused by an inability to use existing work developed with alternate tools also increases costs and reduces the achievable size of models. Ideally, a larger model could be developed from a combination of custom built and pre-existing components (that were themselves created with the tools that most suited the task at hand). These models could then be brought together and work jointly as a distributed simulation. The interoperability benefits of the HLA can help to realise this goal.

How is Interoperability Achieved in the HLA?

Before discussing how the interoperability benefits of the HLA can help the wider business community, it is first important to discuss how the HLA achieves interoperability across otherwise disparate platforms.

Interoperability is achieved within the HLA in a number of ways. Central to all of them is the role of the FOM. Describing the structure of information a given federation can produce and consume as a whole, the FOM is the primary artefact that allows federates to work with one another in a defined manner. A FOM defines the shared vocabulary of a federation. From an interoperability perspective, this means that any simulation component willing to produce or consume information in the same manner can potentially work with other such component (even if they were not originally designed to do so).

In an attempt to gain widespread interoperability among components designed for similar environments, the notion of *reference FOMs* has long been employed within the defence domain. The creation of a central, standard FOM for a specific purpose (such as the Real-time Platform Reference FOM [108]) allows components to be created with interoperability in mind. While such components may not be created with the express intent of interoperability with another given federate, the fact that both support the reference FOM opens this possibility. There is nothing specific to reference FOMs that supports this behaviour beyond the creation of a shared syntax and the implicit agreement of shared semantics (often codified in supporting documentation). Federates designed to work with different models can still be brought together if there exists significant overlap between the models.

Beyond defining what is required of simulation components if they are to work together in a distributed simulation, the FOM also helps overcome platform differences. Assuming very little about the target platform, all communication within the HLA is achieved through the passing of an opaque series of bytes. Where structured data may have different representations on different platforms, a group of bytes as used within the HLA will always remain the same, regardless of platform.

This naturally raises issues when attempting to reconstitute any received information into a format that is both usable and has some sort of semantic meaning. Object models thus form a primary mechanism through which interoperability is achieved. Describing the structure of information that is to be exchanged within a distributed simulation, the FOM acts as a recipe for the reconstitution of received information into its intended format. The exact manner in which this process occurs is dependent on platform and simulation. With the introduction of the IEEE 1516 specification, the OMT format was extended to include all information necessary to define the structure of information [49]. Whether it be simple, primitive data types or large, complex structures, the FOM now contains all the necessary information to both render an opaque series of bytes in some semantically useful form, and to reduce a structure into a series of bytes ready for transmission.

How Interoperability Helps the Wider Business Community

As discussed above, the wide and varied landscape of tools used within the wider business community can have many negative effects on simulation development, costs and return-on-investment. The interoperability benefits of the HLA can help address these problems by allowing models developed across different tools to work together. Development costs can be reduced through the ability to leverage pre-existing simulation components that were previously unusable. Simulations can work together to form larger and more complex models, in turn delivering a greater depth of information and better equipping those who rely on such feedback for decision support. This also enables the reuse of existing component, increasing the return-on-investment made in their development of purchase.

3.4 Summary

While use of the HLA within the wider business community would provide many potential benefits, the application of this technology beyond the domain of defence has thus far been minimal [116]. Despite offering many proven interoperability benefits, the HLA still presents many sizeable shortcomings that must be addressed before it can be readily employ beyond the defence community. Although the IEEE 1516 standard is currently undergoing periodical revision and is addressing some of these shortcomings, many problems still remain. This chapter has introduced the HLA, its processes and components. Chapter 4 discusses the shortcomings of the HLA as it is specified, and outlines some of the research and development efforts made to address them.

Chapter 4

The HLA: Problems and Solutions

A proven, international open standard, the HLA has the potential to enable increased interoperability and reuse of simulations within the wider business community, in turn lowering development costs, increasing return-on-investment and allowing larger and more encompassing models to be developed. However, while capable of delivering considerable advantages, the HLA as specified presents numerous sizeable problems. This chapter introduces and discusses these issues, before outlining the research and development efforts that have attempted to address them.

4.1 Shortcomings of the HLA

While the HLA presents significant advantages when considering it for use within the wider business community, there are a number of shortcomings that currently prevent any wider uptake of the technology beyond the defence domain. The problems of the HLA are well known and have been the subject of much work within the community.

4.1.1 Development Complexity

Development complexity is perhaps the single largest noted deficiency of the HLA. Having been the focus of considerable attention and research, these issues have been widely identified as increasing the time and costs involved in distributed simulation development.

Access to the simulation services offered by the RTI is only available through the low-level facilities defined in the interface specification which area complex and difficult to work with [20]. In use, these interfaces can be unintuitive and require expert knowledge to use effectively. While quite powerful and flexible, the interface specification causes many problems at an implementation level.

<u>Code-Space Inefficient</u>

The application code required to utilize the HLA services is extremely code-space inefficient, often requiring hundreds of lines in order to achieve simple outcomes. The process of programming a federate is a complex and laborious task [82]. Further, the asynchronous programming model used by the HLA requires that the client manage the association between event responses and their original requests. This approach results in the need for an additional coding effort and in turn increases the amount of code that must be developed and maintained [83].

Unstructured Development

The low-level nature of the facilities provided by the interface specification means there is no consistent, accepted model driving their use. As such, simulation developers often resort to the development of their own infrastructure and middleware solutions in order to reduce the development effort required [83]. This in turn results in a duplication of effort and creates an additional source of development and maintenance. Further, the process of testing and debugging federations is recognized as a difficult and time-consuming process [84].

Code Tangling and Tight Coupling

Without a formal standard defining the proper separation of model code from the integration logic required to support the RTI ambassador interface, the two often become entangled and are unable to be considered separately. Such *tight coupling* of business logic (in this case, a simulation model) and underlying support code is widely recognised as a major factor in constraining reuse potential.

Code required to communicate with the RTI and handle incoming information ends up scattered among the pure logic of the simulation, making models more difficult to comprehend and creating maintenance issues [92]. This entanglement harms model comprehension and can make it more difficult to spot errors or bugs. The tight coupling that exists between model logic and the underlying HLA infrastructure code also makes it difficult to reuse that effort in situations where the infrastructure must be changed.

Data Interpretation

The IEEE 1516 OMT standard defines a format that allows object models to fully specify their contained types. This information can be used to construct properly formatted data from the opaque series of bytes received from the RTI. However, the process of serialization and de-serialization is left entirely to a federate developer. While this is a simple process for primitive data, when considering larger data structures, it can become complex and burdensome.

The simulation specific services provided by the HLA require expert skills and training not generally found in the wider business community. While general programming knowledge is widespread and pervasive within such domains, the knowledge required for HLA federation development is not. Despite being available as an internationally defined open standard for numerous years, the HLA has yet to gain traction beyond the defence domain. Given this minimal uptake, it is clear that the skills required, and the complexity involved with the HLA as it currently exists, render is undesirable in such contexts.

4.1.2 Interoperability and Reuse Shortfalls

A major strength of the HLA is the interoperability benefits it provides. Linking simulations and components on diverse and disparate platforms together, the HLA enables the interoperation of many different simulation styles. Central to the interoperability benefits of the HLA is the notion of a shared object model (the FOM). However, while successful in enabling a broad level of interoperability and reuse, there are situations in which the HLA falls short of its goals. Most commonly, HLA interoperability breaks down when attempting to use a particular federate in a context it was not specifically designed for.

Syntax v. Semantics

Defining the shared vocabulary of a particular federation, any federate that conforms to a given FOM can work in the same distributed simulation as other such federates. However, the FOM only defines the *syntax* for interoperability, not the *semantics* [112]. While direct conformance with a FOM is the first step towards interoperability, there are many situations in which semantic differences in the behaviour of a federate lead to incompatibilities. Broadly speaking, these can be broken down into three distinct categories:

Execution Management

Differences relating to the way a federate may manage its execution within a federation. Different federations have different execution processes. Some use synchronization points to tell federates when to start and stop execution, others may use interactions. Certain federations may have defined points at which they publish and subscribe or pre-register all their object instances. In order to enable repeating simulation runs without exiting, the Virtual Maritime Systems Architecture (VMSA) simulation mandates that a federate must go through a simulation state save and restore process before each run [11]. If a given federate can not conform to the execution management procedures expected by other federates, the simulation may never be able to start.

Model Differences

Model differences relate to the way a given simulation component may produce or consume information, and the assumptions it makes about simulation data. Two models may produce information about the altitude of an aircraft as a 32-bit integer, but one interprets it as meters while the other as feet. While the OMT supports the definition of such information in the FOM, it is formed as free text and cannot be processed directly by a computer. As such, to adapt one federate to a new FOM would require code-level changes.

Problems can also arise when federates do not agree on policies relating to the items such as update production rates. While some federates may attempt to reduce bandwidth by sending infrequent updates (relying on dead-reckoning to make assumptions in the mean time – as was popular with DIS), other federates may depend on a continual stream of information.

HLA Service Usage

The interoperation of federates depends heavily on the HLA services they may use of and depend on. While two federates may be compatible at an execution and model level, if one depends on co-operative modelling through the transfer of attribute ownership and the other does not, significant problems arise. Common areas of concern relate to models that do or do not depend on ownership or time management.

FOM Agility

Compliance with a particular FOM is no guarantee that a given federate can interoperate with another compliant federate. Interoperability within the HLA is dependent on the FOM acting as a shared contract describing the data (and structure) federates wish to use. *FOM Agility* is concerned with the processes involved when a significant functional overlap exists, yet the way data is represented within the FOM is different [40]. In order to invoke RTI ambassador services, FOM-specific information must be known. Information such as object class and attribute names are often hard-coded into a federate, and as such, switching to a different FOM necessitates alteration of a federate at this level. In situations where this is not possible, the ability to reuse such a federate is lost.

The FOM-centric approach of the HLA leads to issues of cross-federation interoperability and significantly reduces the ability to reuse existing work in a context for which it was not originally intended. The use of reference FOMs is seen as one way to avoid this problem (by standardising on a specific model). However, this workaround does not solve the underlying problem. Further, it is only of benefit when all use of a given federate is intended to be with regard to the same model.

Link Compatibility

While FOM-Agility and semantic differences are classes of interoperability problems that relate directly to the underlying model, the issue of link compatibility is a problem the infrastructure used to enable a distributed simulation.

To allow unprecedented flexibility with regard to the way HLA types[3] are represented within a specific RTI implementation, the IEEE 1516 interface specification mandated a set of abstract, vendor neutral types [48]. This allows a vendor to implement such types in any manner they desire, while insulating federates from the underlying semantics. However, sufficient means of creating and obtaining these types was not supplied, requiring federate developers to make direct use of vendor-specific types. This in turn required source code alteration and recompilation when attempting to move a federate to a different RTI implementation [43].

The lack of link compatibility further reduces the interoperability and reuse potential of existing federates. The dependence on a specific RTI means a federate can only be used in situations where the implementation it was designed for is also in use. To address this problem, the Simulation Interoperability and Standards Organization (SISO) developed the companion "Dynamic Link Compatibility" specification (otherwise known as the DLC) [41]. The focus of the DLC is to remove these anomalies and enable a smooth transition when moving from one RTI implementation to another [45]. However, while the DLC specification does address many of the situations in which link compatibility problems arise, in its current form it fails to realize this goal with regard to LogicalTime types.

Standards Compatibility

Having been in use for many years, the HLA has undergone several specification alterations and upgrades. At the time of writing, there currently exist several different versions[4]:

- HLA 1.3 (initial DMSO version)
- IEEE 1516
- SISO DLC

[3] Types such as LogicalTime implementations, abstract handles that identified a specific object instance, class, or attribute (for example).

[4] The HLA Evolved has come about as part of the general process of refreshing IEEE standards after a certain period of time. At the time of writing it is about to be voted on and release is anticipated in late 2006.

The DLC standard was an extension of IEEE 1516 to address link compatibility issues. As such, the two are mostly compatible. However, the changes to the interface specification between HLA 1.3 and IEEE 1516 were significant.

While the evolution of a standard is necessary to keep it current and avoid stagnation, in the case of the HLA, backwards compatibility has not been preserved. Incompatible alterations to the interface specification have been made such that alteration of a federate at a code level is required when attempting to work with different versions. Especially vexing is the choice to change the names of various types; even when there is no semantic difference in the way they are used. This is further exacerbated by changes made to the names of object, attribute and interaction classes. As these types must be referenced by name within a federate, such an action immediately invalidates all federates of the previous versions, excluding them from operating correctly.

As the standards are no longer compatible, in order to maintain a federate capable of working in either situation, two separate lines of development must be created. While adding additional maintenance costs, these actions reduce the reuse and return-on-investment potential of a simulation component.

4.1.3 Barrier to Entry

Rather than being a single problem that restricts the broader uptake of the HLA, barrier to entry problems are the product of many shortcomings. The current environment surrounding the HLA actively inhibits an expansion of the community into the mainstream business environment.

Infrastructure Costs and Open Source Software

While common in the broader business community, the HLA has a distinct lack of Open Source Software (OSS) involvement. While in and of itself this is not necessarily a negative, the prohibitive costs of commercial infrastructure and support tools serves to magnify any issues it may raise. In relative terms, compared with general simulation tools, the HLA is a niche market. As is common in such situations, commercial infrastructure and tools are expensive.

RTI software is essential for HLA and without it, the development and execution of simulations cannot occur. While robust and mature commercial offering are available at considerable cost, the lack of any basic tools that can help expand use of the HLA is noticeably absent. Without such options, those exploring the potential of the HLA to help solve their problems have no support in assessing and evaluating its advantages without the involvement of significant financial commitment. Further, the expense involved in commercial offering also restricts the ability of small-to-medium enterprises to compete with larger competitors.

Up until December 2002, the US Defense Modeling and Simulation Office (DMSO) made their RTI implementation widely available (subject only to International Traffic in Arms Regulations - ITAR). With this contribution, anyone interested in learning or assessing the HLA had at least the minimum required tools at their disposal. However, citing a growing commercial RTI market, DMSO removed their offering from public availability [52], leaving no non-commercial options available. With no freely available option, the already significant barrier to entry is raised yet again.

The need for publicly available tools has not gone unnoticed and the potential of Open Source Software (OSS) has been identified as a potential solution [113]. While pervasive in the mainstream community, OSS has yet to make any significant inroads into the HLA community. However, recent times have seen a number of projects begin to emerge [1, 74, 93]. Despite being early in development or relatively incomplete when compared to mature, commercial offerings, these endeavours hold considerable potential for addressing the lack of free tools. Reducing the barrier to entry, they may be able to help grow the community and in time feed into the market for more mature, commercial options. With this in mind, the general availability of basic tools is becoming a much smaller hurdle.

Despite offering many advantages for the development of distributed simulations, the environment described above significantly restricts a broader uptake of the technology. These problems arise directly from the requirements of the HLA as specified and group together to make the prospect of HLA use both excessively complex and costly.

As the expression of a model within the HLA is so inherently tied to the directly exploitation of RTI services, specialist knowledge and training is required (and the learning curve is steep). The complexities involved in HLA simulation development in addition to the considerable amount of code that must be written leads to longer initial development timelines and increased maintenance costs. Shortcomings in the ability to reuse existing work lead to an increased duplication of effort, resulting in additional cost and time commitments. The lack of model composability caused by these reuse problems reduces any capacity for the rapid development of even small models.

Taken together, all these issues combine to vastly reduce the attractiveness of the HLA despite the potential benefits if can offer. The high barrier to entry is the combination of all the problems discussed in this section and serves to restrict a broader uptake of the HLA within the wider business community.

4.1.4 Summary

Section 4.1 has introduced the common problems associated with HLA simulation development. The issues of development cost and complexity, a requirement of expert skills and training, and interoperability and reuse shortcomings have been highlighted as the major deficiencies of the HLA as specified. When considered alone, these failings render the HLA unsuitable for use within the wider business community; considerable research and development effort in recent times has focused on solving the issues. Having identified the primary shortfalls of the HLA, this section has provided the background for a discussion on recent research and development efforts aimed at addressing the problems.

4.2 Addressing the Problems

The problems of the HLA are well known within the research and development community. While the HLA has long enjoyed pervasive use within the defence domain despite these issues, compelling solutions must be found before the HLA can be made fit for use in a mainstream simulation context. This section introduces and discusses some of the research and advancements aimed at addressing the shortcomings, their successes and failures.

4.2.1 FOM Agile Federates

The term FOM Agile federates refers to a concept that involves creating mappings to describe the transformation between the model that a particular federate requires and that which is being used in the wider federation [124].

As the process of moving a federate to a new federation can involve both semantic and vocabulary differences, some changes need to be applied to the federates before they can work in their new setting. One option is to modify the source code for the federate directly. This approach, while effective, requires a user to come to terms with not only the model that the federate uses, but also the way in which the federate was developed. The amount of time it can take to port a particular federate can be significant, and as such, costly. Further, once a federate has been ported to a new federation object model, two separate versions of the federate now exist. This raises maintenance concerns and again requires significant investments of time and money.

The FOM Agile approach seeks to use middleware as a kind of Rosetta Store. The middleware can be configured to perform transformations both in terms of simple naming changes, right the way up to complex data transformations [40]. By placing middleware between the federate and the RTI, these types of transformations can be performed, essentially fooling federate into believing that the federation is communicating in terms it natively understands.

The problems of FOM Agility have been the focus of considerable research and development effort in recent times. Through the ideas and concepts presented here, many of the issues can be mitigated. Any solution that intends to offer simulation services to the wider business community must provide the ability to integrate such approaches so that a fuller picture of interoperability and reuse can be realised. As part of the periodic revisions to the IEEE 1516 standard, the notion of "Modular FOMs" is being introduced. This allows individual federates to describe only the parts of the FOM they have interest in. The RTI in turn merges all the constitute modules from all the federates into the actual federation model. This particular enhancement is one such effort that is being made to allow federates to become more agile, letting them focus purely on the parts of the model of interest to them. Although it helps, it does not provide a solution to problems of agility when modules overlap and make use of different semantics. In this situation, more traditional mapping and transformation techniques are still required.

4.2.2 Code Generation

Section 4.1 identified the problems of development cost and complexity as significant factors holding the further expansion of the HLA back. By automating the creation of large portions of a simulation component, code generation can help to reduce development time and costs. Unlike the other technologies presented in this section, code generation is not a general solution that is used in isolation. Rather, such techniques normally form part of a larger overall strategy (as will be discussed in relevant parts of section 4.2).

Code Generation Overview

A code generator is a program that produces other programs [51]. The basic process of code generation involves the automated conversion of a high level description of a piece of logic or entity into the actual code required to implement it. There are many advantages to using this process [70]:

- Reduction in the amount of code which must be authored by hand

- Improved code quality as output is automatically generated, not written by hand (and subject to the mistake people make)
- Reduction in maintenance costs as local at which bugs are produced is centralised. If there is a bug, it is everywhere code generation was used and is thus easier to find.

There are two general types of code generators, each of which is discussed below:

Binary Runtime Code Generator

A runtime code generator creates code dynamically during the execution of a program. These types of generators work at a low level and are generally not exposed to a developer (they are hidden in an execution environment) [51]. Such code generators create code for use during a particular execution, with the code produced not persisting beyond the runtime of the application. Runtime generators can be found in places like the Java Virtual Machine [115]. Here, Java byte-code is turned into machine executable code at runtime.

Source Code Generators

As opposed to runtime code generators, these types emit actual source code [51]. Used in many environments that provide application development support, source code generators create code on behalf of a user. Used to generate code for common, repetitive or monotonous tasks, the code created by this type of generator is intended to be persistent, and often times is extended or directly leveraged by a developer. Many applications make use of this type of generator, examples of which include Rational Rose, Microsoft Visual Studio and Calytrix SIMplicity (a HLA simulation development tool discussed in section 4.2.5).

Successes and Failures

In situations where significant amounts of repetitive code must be written (such as the HLA), code generation can help to reduce the burden on developers. As the volume of code produced is decreased, so to does development timelines and costs. However, in and of itself, code generation does not solve many problems of the wider business community. Generally, code generation is a low-level technology that is leveraged in the context of a broader solution, such as those discussed next.

4.2.3 Component Models

Considered an important solution to many of the common problems faced in software development, component models are credited with improving both productivity and code quality [55]. Through the use of component based development many organizations have claimed significant benefits from the increased reuse and interoperability they offer [44]. Component based development involves the process of separating the core business logic from the platform that it aims to exploit [46]. Separating the business logic from the underlying integration code allows for either to be changed without the need for extensive redevelopment, resulting in the realisation of reuse and interoperability across differing platforms.

A component model is an architecture that allows application developers to define reusable fragments that can be combined to create a larger application [87]. These fragments are known as components and they form the basic building blocks through which applications are assembled. Behind the abstractions provided by a component framework is the *container*. The container provides the concrete environment in which components execute.

Components

A component is a self-contained, self-describing unit of functionality [84] that is deployed in a managed environment. Components themselves do not necessarily constitute entire applications; rather, they provide small pieces of logic that may be aggregated to form an application [97].

A component is composed of two parts: the logic it implements (its interface) and the description it provides of itself. The interface provides programmatic access to the internal state of the component and is provided through the use of abstractions defined by the framework. Metadata provides information about the component, such as vendor and version descriptions in addition to the container services it wishes to leverage [84].

Rather then acting as a standalone piece of software, components are deployed into a container that provides an execution environment. A component contains the core or business logic required to complete a task. Rather then combining the code required to tie this logic into the execution environment a component interacts with its container through an abstracted interface. Separating the business logic from the underlying platform details allows a component to be deployed into any container that conforms to the component model [83].

Component Containers

The middleware services required by a component are collectively described as the component container [84]. A component container provides a concrete implementation of the abstract interface to which the components conform and may provide access to a number of services that are transparently applied to deployed components, such as [84, 87, 97]:

- Transactional Services
- Naming Services
- Security Services
- Messaging Services
- Persistence Services
- Quality of Service (QoS) Services

The abstraction of the actual container implementation from the business logic contained in the component allows for the drop-in replacement of either the component or the container (thus providing a substantial increase in the level of reuse and interoperability possible).

Component Based Development

Component based development advocates the partitioning of the application logic into smaller, reusable components [6] that communicate with a platform through an abstracted interface. An application is logically partitioned into a number of software components where each of these components conforms to a common component model [84].

Given that the actual platform implementation details of the container in which components execute remains hidden, it becomes possible to switch implementations of this container without the redevelopment of the application logic (which would otherwise be required). This same principle applies for components. Since components are self-contained units with shared assumptions regarding the manner in which they interact and connect with their executing environment, they can be swapped for other components that provide the same level of functionality [39]. Additionally, the specification of a standard interface allows components to remain portable across many physical container implementations.

While increasing the reuse potential of developed components, this process also helps to increase developer productivity by removing tedious tasks from the process. Further, a significant reduction in maintenance can be achieved through the reduction in developed code. With the application logic partitioned into self-contained components it is possible to develop and maintain each component separately. Beyond this, the use of standard supporting infrastructure can significantly reduce the amount of code that must be authored to provide communication between components [3], easing the development burden associated with inserting new functionality into an application. From this description we can see that the primary goals of component-based development are [46, 84]:

- to develop software from pre-existing parts
- to reuse these parts in other applications
- to easily maintain and customise these parts to produce new functions and features

Successes and Failures

The HLA was developed by the US DoD in order to increase the reuse and interoperability of distributed simulations; however, although significant advances have been made, the full realisation of this goal has thus far remained unachieved. As discussed in the section 4.1, issues such as FOM Agility and the tight coupling of simulation logic and infrastructure code all impede the ability of HLA simulations to achieve the level of reuse and interoperability desired [42, 63, 83]. The application of component-based development methods has the potential to help rectify this situation and realise the goals of reuse and interoperability.

The Simulation Component Model (SCM) has been developed specifically to address the needs of the HLA. The SCM applies component-based development techniques to the HLA in an effort to realise greater reuse and interoperability of the core simulation model logic involved.

The Simulation Component Model (SCM) provides support for the development of reusable HLA based simulation components [83]. While the HLA provides a solid base for composing distributed simulations in a reusable manner, realising this goal is difficult to achieve. The SCM applies component based development techniques to the HLA in an effort to realise greater reuse and interoperability of the core simulation logic involved in a simulation. Reproduced from [83], Figure 4-1 shows a top-level view of the SCM and the component/container relationship:

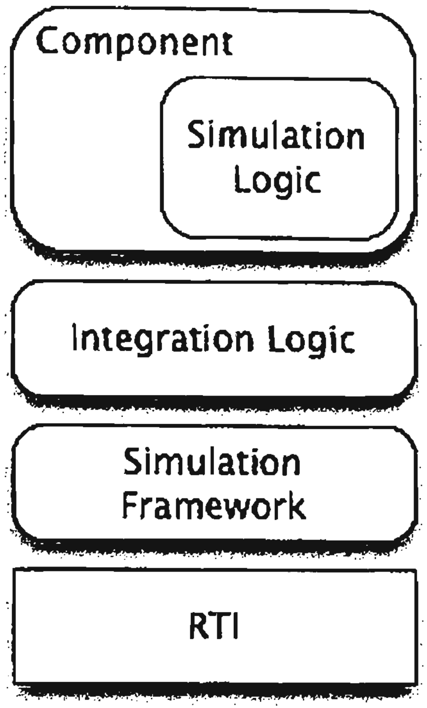


Figure 4-1: Simulation Component Model [CC]

As shown above, the SCM separates model logic into reusable components and defines a container in which they can be deployed and access the hidden, underlying HLA services. Building on previous work, the integration logic required to connect the component to the simulation framework can be automatically code generated from metadata information.

Component models such as the SCM offer many advantages with regard to the development time and costs involved with the creation of HLA-based simulations. Through their connection to the HLA, considerable interoperability and reuse potential is also possible. Through the execution container, an SCM component appears just like any other federate within a federation. It may interoperate with other SCM-based components or federates that are built directly on the HLA interface. As such, the reuse of existing components in new works is entirely possible. To overcome FOM agility issues, the container can transparently use facilities such as those described earlier.

Despite offering many compelling advantages with regard to the development, interoperability and reuse of simulations, component models such as the SCM still depend on specialist HLA knowledge not present within the wider business community. A more general solution that abstracts the HLA itself is required to help realise an enhanced uptake of HLA-based distributed simulation.

While this shortfall means that component models such as the SCM do not themselves provide an environment entirely suited to the problems of the wider business community, such techniques can find use within the underlying implementation of a more general abstraction. Just as component models hide infrastructure services, a more generalised approach is needed to hide HLA knowledge (and the complexities involved) from an end user.

4.2.4 Middleware

Much like component models, Middleware solutions are designed to substantially improve developer productivity through the abstraction of low-level, tedious and repetitive tasks. While component models could be viewed as a form of middleware, generally speaking, component models tend to provide more advanced services transparently through the container. Unlike component models, the middleware approach is generally more lightweight and the focus (within the HLA) is still on developing federates as opposed to reusable components. One example of this is the pSISA (Proposed Standardized Interface for Simulation Applications) middleware, which places various layers between underlying simulation logic and raw HLA in an attempt to abstract the complexities and provide additional useful services [69]. Another example is *ProtoCore*. Created by the US Army as a unifying API that simulations could be developed, the goal of ProtoCore is to again remove the specifics of the underlying distributed simulation technology from the simulation itself. This in turn would allow simulations to be quickly and easily deployed with newer versions of the HLA, or, in the case of ProtoCore, other simulation technologies such as DIS [109].

Successes and Failures

Given the close alignment between component models and middleware solutions, the successes and failures of each are virtually identical. Frameworks such as Middlesim [93] allow users to achieve increased productivity through the abstraction of the HLA interface. While Middlesim does not incorporate such facilities itself, FOM agility issues can also be overcome through the application of techniques described in section 4.2.1. Given this, middleware based simulations can provide much the same level of interoperability and reuse benefits as component models.

However, while displaying the same typical gains in development time, interoperability and reuse as component models, middleware solutions also present the same drawbacks. With regard to this work, it is the requirement of specialist HLA knowledge that reduces their effectiveness in the context of the wider business community.

4.2.5 Tools Support

Throughout the maturation period of all successful infrastructure standards, a key development has been the production of tools that simplify their use [49]. The HLA provides a strong foundation for distributed simulation development; however, it is an extremely complex framework to use and would benefit considerably from tools designed to support the development process. The notion of an Integrated Development Environment (IDE) supports this idea. An IDE is designed to automate or hide many of the tasks found in every development effort. Wizards and visualisations of certain problems present a user-friendly interface that the IDE can use to generate large amounts of code and configuration data on behalf of the user.

Tools Support in the HLA

In the context of HLA-based distributed simulation, such tools generally fall into one of two categories: generic or FOM-specific. FOM-specific tools tend to focus on the Real-time Platform Reference model. As the object model is consistent across all situations, tools such as OneSAF [128] tend to focus on the composition of entities and scenarios from pre-built components rather than the creation and coding of those entities. The ability to drag and drop together a simulation is a clear benefit both in terms of usability and productivity. However, given the specialised nature of these tools, they are of little use beyond their context.

On the other hand, generic applications provide an environment free from such constraints. The SIMplicity [86] IDE currently represents the state of the art with regard to such HLA-based simulation development tools. SIMplicity is a commercial product developed to support the HLA development process. The SIMplicity environment embodies many qualities that ease the burden of simulation development and reduce the time and effort involved, help realise increased simulation interoperability and aid in the process of deploying and executing a distributed simulation.

SIMplicity relies heavily on the use of visual tools and automated code generation to help abstract many of the underlying problems of the HLA. Graphical interfaces are used to compose and visualise many HLA development artefacts or processes such as:

- **FOM Development:** Object models can be composed through the use of standard diagramming techniques [16]
- **Federate Development:** The publication and subscription interests of a set of federates is described via a custom diagramming process [17]

- **FOM Mapping and Agility:** Mapping rules can be visually composed to help bridge the gap between the object model of a given federate and that of the federation it is to participate in [18]
- **Deployment and Execution:** Graphical interfaces are used to aid the process to deploying a simulation component to a given computing resource and to centrally control its execution [19]

Through the use of code generation, SIMplicity is able to produce a large amount of infrastructure code on behalf of a developer. Once the simulation model has been specified, the code generation process can begin [86]. In order to provide a structured process for the development of distributed simulations, the code generated by SIMplicity conforms to the SCM discussed in the previous section (the SCM was initially developed as part of SIMplicity). This provision supports the clean separation of simulation logic and infrastructure code and simplifies the process of developing a HLA based simulation. Following code generation, a developer is tasked with the role of authoring model code required to complete a federate.

Successes and Failure

The tools and processes put in place by the SIMplicity development environment considerably reduce the effort required to create a HLA simulation. Through the use of the SCM developers are insulated from the low-level details of the HLA and are able to rapidly create simulation components. However, despite the many advantages SIMplicity provides, knowledge of the HLA is still required when filling out the code that it generates. In the context of this research, the requirement of such knowledge must be further removed.

4.2.6 Migration of HLA Services to Civilian Applications

One of the problems with the HLA is that its implementation infiltrates a simulation modeller's entire development process. Rather than providing optional support for simulation distribution, use of the HLA necessitates that it be considered during every step of the development process. One natural way to reduce complexity would be to allow simulation modellers to optionally access HLA distribution services as just another function of their chosen modelling environment. Grafting the HLA onto applications used within the wider business community for modelling purposes would allow access to those services from a setting in which modellers are already comfortable.

The notion of coupling HLA-based distributed simulation with tools commonly found beyond the domain of defence is not new. In [112] Straßburger explored approaches for applying the HLA within "civilian" simulation applications. Investigating the similarities and differences between the two communities, [112] established that the HLA could provide benefits to in the mainstream domain (citing interoperability as a primary concern). Accordingly, a major focus of that work was the development of exemplars that combined the HLA and common simulation development applications.

Straßburger identified many requirements for the connection of HLA and civilian simulation tools. When considering how HLA simulation services would be accessed from within such tools, the notion of implicit and explicit access was presented.

Implicit and Explicit Access

When attempting to allow use of the HLA through mainstream simulation tools, consideration of how a user will access HLA simulation services is paramount. To this end, Straßburger identified that there are two general categories: implicit and explicit access.

Explicit access defines all situations where a user has direct contact with the HLA, much as a

> *"It is especially interesting to note that the implicit approach, which in the opinion of the author is the best approach for hiding HLA functionality from the user, has not been implemented by anybody else. This is very unfortunate, since this approach requires the least user involvement for building HLA federates. For avoiding the symptoms of the parallel simulation community in terms of lacking impact of the general simulation community, the implicit approach seems to be the best solution. It requires no adoption of new modeling paradigms, world views, etc."*

regular federate developer would. Simulation services are invoked manually, and as such, this approach requires knowledge of the HLA and how to use it. In many ways, explicit access is akin to authoring a federate manually; with the major benefit being that the environment used to develop a pure model is the same as the environment used to develop its distributed version. While the development tool may be different (and able to provide extended services), the explicit approach still necessitates manually supporting the RTI.

Conversely, the implicit approach focuses on the abstraction of the HLA behind the tool-specific view of a simulation world. Simulations are developed in the manner applicable to the tool in question. From the perspective of a user, underlying support for the HLA is transparent. This approach naturally does not require the intimate knowledge of the HLA that is necessary when manually authoring a federate, of making use of a tool that embodies the explicit approach.

While the explicit approach provides easy access to the HLA from mainstream tools, a requirement of HLA knowledge renders it unsuitable in the context of this work. However, the **implicit** approach described by Straßburger raises many potential benefits with regard to the goals of this research. In [112], Straßburger takes the first real steps towards the use of HLA within the wider business community. His dissertation investigates, compares and contrasts the explicit methods, highlighting the strengths and weaknesses of each. The primary experimentation carried out involved the grafting of the HLA onto existing simulation modelling tools. For each tool, a specific interface to the HLA was defined (each falling into either the explicit or implicit category).

In his concluding remarks, Straßburger notes [112]:

Where Straßburger demonstrated how the HLA and civilian simulation tools could be combined through a group of tool-specific interfaces, the goals of this research are to extend the implicit approach into a generalised, non tool-specific context. Building on [112], this research seeks a general method that allows pure models to be rendered as HLA simulation components.

In order to realise these goals, facilities that can be used to separate the process of developing a pure model from that of enabling it as a distributed simulation component are necessary. While capable of reducing the complexity involved in working with the HLA, the technologies presented thus far in this section still require some specialist knowledge. As such, the use as an underlying platform to support this research is not possible. Facilities that allow for the complete abstraction of HLA semantics are needed. The next two sections introduce two technologies that have been the subject of significant amounts of recent research and have the potential to help realise the goals of this work.

The Model Driven Architecture (MDA) has been the focus of much research within the modelling and simulation community in recent times. Applying an implicit approach to system development, the MDA process advocates raising the level of abstraction such that a model is created free from *any* implementation information. Additionally, these models are specified in a visual modelling language. The richness of a visual medium theoretically allows for simpler comprehension of how a model fits together and helps to speed its development. Before deployment, these models are transformed into an executable implementation. Section 4.3 introduces and discusses the MDA.

Aspect-Oriented Programming (AOP) is a slightly different approach to systems development. The primary focus of AOP is on the *separation-of-concerns*, whereby the implementation of business logic and that of implementation logic is kept separate. This is very similar to the MDA approach, however, rather than relying on a largely visual approach, AOP operates at the programming language level. Section 4.4 introduces and discusses AOP.

4.3 The Model Driven Architecture

The focus of considerable attention and research in recent times, the MDA has been suggested as a potential solution to many of the problems facing simulation development. Developed and standardised by the OMG, the MDA is an attempt to dramatically simplify the development complex systems through a focus on modelling rather than code authoring. Utilising the Unified Modelling Language (UML), systems can be composed via a largely graphical process, focusing on the pure model rather than the underlying infrastructure platform. Through an enhanced focus on the problem, insulation from changes (or the complexities) of the underlying implementation platform is provided.

Many advantages have been claimed by the MDA relating to decreased development time and complexity, increased reuse and interoperability. While the MDA vision represents a substantial advancement, its full realisation has thus far proven difficult [90]. This section investigates the processes involved.

4.3.1 MDA Overview

The Model Driven Architecture is described by the OMG as their next step in ensuring interoperability, portability and reusability [110]. Moving the focus of a developer or architect away from the technology on which a solution will eventually be implemented and onto the core business problem, the MDA is a new way to specify software [110]. In order to provide an open, vendor neutral approach, the MDA has been based on OMG modelling standards such as the Unified Modelling Language (UML) and the eXtensible Metadata Interchange (XMI) [75] (which describes how a UML model can be serialized and accessed in a structured way). The processes that make up the MDA can be split into three distinct sections, broadly concerning three different representations of a system. These processes and representations are discussed below.

The Platform Independent Model

Under the MDA development process, a model of the core problem or system becomes the central artefact of the development effort. This model (known as a **Platform Independent Model** or PIM) is intended to define the pure problem, containing no reference to any platform or implementation technology. Through the removal of such considerations, the MDA allows developers and architects to focus solely on the business processes and the behaviour of the actual underlying system, rather then having to simultaneously deal with complex technological considerations [85].

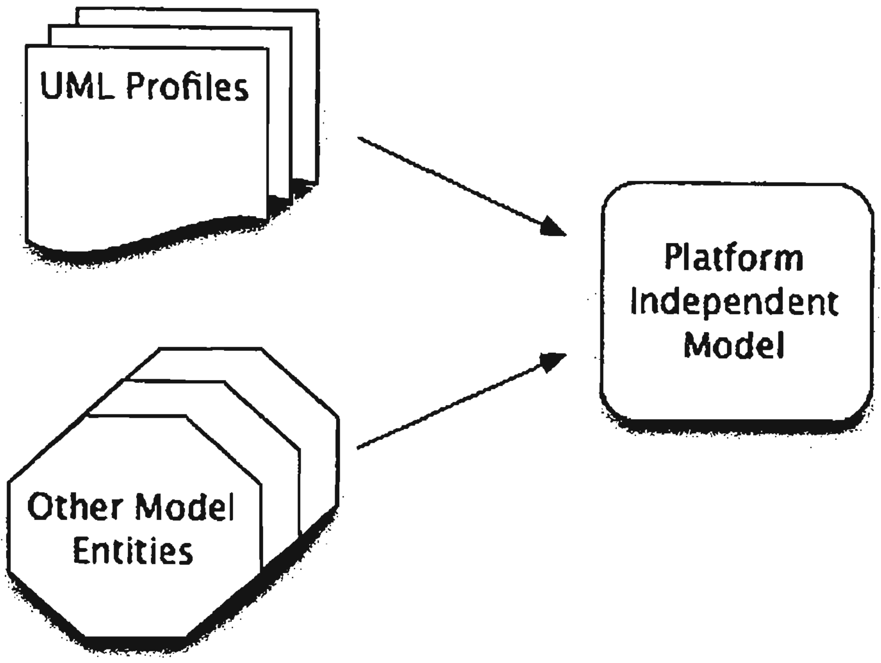


Figure 4-2: Development of a PIM

In line with the goals of the OMG to base the MDA around open, industry standards, a PIM is defined using the Unified Modelling Language. The use of UML enables the ability to leverage UML profiles during the development of a PIM. A UML profile is an extension of the basic UML standard to describe various domain specific entities [77]. For example, a Financial Modelling UML profile might define a "cash flow" entity, its properties and behaviour. The use of UML profiles allows PIM developers to rapidly create their model using predefined, standardised entities.

The Platform Specific Model

Once the pure problem has been modelled independent of any platform complexities or details, it must then be transformed into a **Platform Specific Model** (PSM). Encapsulating a shift in emphasis from the business aspects of a system to its technical issues [85], a PSM is in effect a "redrawing" of the PIM to include implementation details. Put another way, *while the PIM defines the necessary functionality, the PSM specifies how this functionality is realised on a specific platform* [117].

Through standardised transformation rules, a PSM is to be derived from a PIM [116]. The use of automated tools is intended to complete this process, removing the need to make a manual transformation from one model to the other. While a manual transformation would necessitate an intimate knowledge of the target platform and would require a substantial amount of time, an automated process would remove this requirement, thus providing obvious productivity benefits [105]. However, while the ideal of an entirely automated conversion process is appealing not only in terms of productivity, but also in terms of cost and time-to-market factors, it is recognised that currently, even the most advanced MDA tools are not able to realise this goal; instead "arguably leaving the most complex considerations to the programmer" [85].

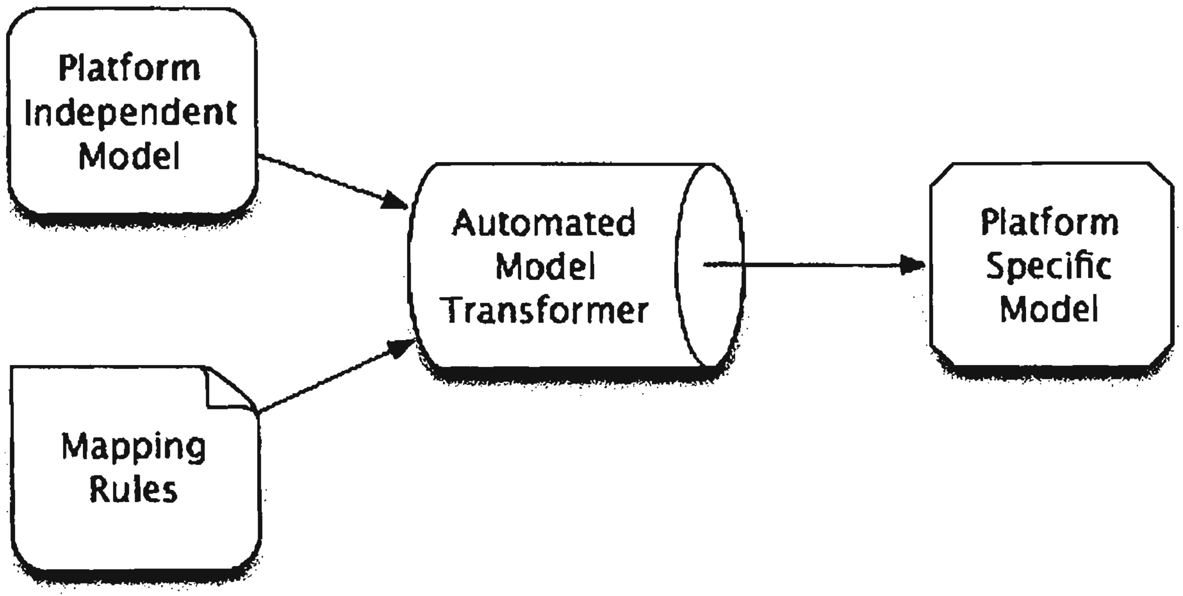


Figure 4-3: PIM to PSM Conversion

As with the PIM, a PSM is a UML model. However, unlike the PIM, the PSM is expected to capture implementation details in a *satisfactory level of detail* so as to enable an automated transition to the next stage. Developments such as the Action Semantics specification (ASL) [78] and Object Constraint Language (OCL) [68] have been created to allow the definition of behaviour within or between entities more concisely than is possible with pure graphical diagramming.

<u>The Implementation</u>

With the creation of the PSM as a rendering of a model for a target platform, the next step in the process is to convert the PSM into an actual implementation. Again, as the PSM is implemented in UML, automated tools are able to access and manipulate it, in this case generating program code (perhaps in a 3rd generation language such as Java or C++) and other required artefacts to enable its deployment. This is the final step in enabling the MDA process. With the ability to generate a complete implementation, the need to author code by hand is replaced with an automated process.

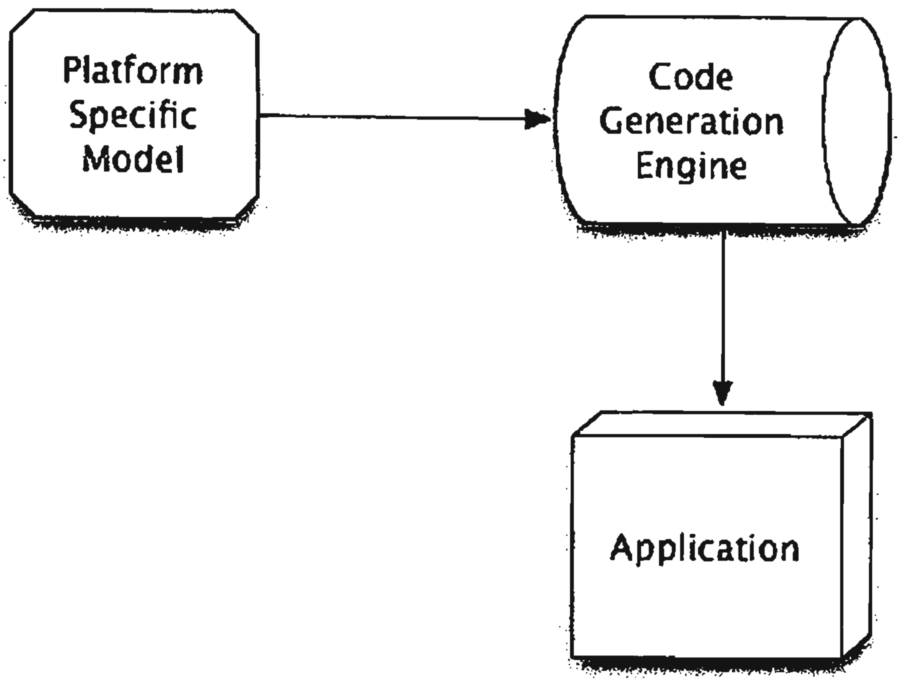


Figure 4-4: PSM to Implementation Conversion

However, as with the PIM-to-PSM transformations, even the most advanced MDA tools are not able to fulfil this goal entirely [86]. While much code can be generated from a functionally complete platform specific model, a significant amount must be manually added in order to produce a complete, executable, deployable component or application.

4.3.2 Advantages and Proposed Successes of the MDA

The MDA process defined by the OMG proposes many advantages, including a reduction in development cost and complexity and an increase in interoperability and reuse. Through a process that embodies the implicit approach also presented in Straßburger's work, the MDA raises many potential benefits.

Focus on the core problem

Through the PIM, the MDA prescribes a process that narrows the initial focus of development to the pure problem, removing the distraction and complexity of the underlying implementation technologies. One of the main contributors to the high probability of failure in large projects [79] is the inability to develop an adequate solution for the user requirements. Through a focus on modelling *only* the core problem, a given solution is much easier to comprehend and the ability to identify deficiencies in a design is enhanced.

Interoperability and Reuse

The enhancements to interoperability and reuse are perhaps the most promoted advantages of the MDA. The information presented here outlines how the OMG describes its function.

Under the MDA environment, reuse exists at many levels:

- Reuse of entities and data types from a PIM in other PIMs
- Use of UML profile entities and data types in many PIMs
- Reuse of standardised mapping rules across many models
- Reuse of a given PIM as the model for many differing PSMs and implementations

The MDA supports the reuse of predefined model entities and types within a PIM through the specification of UML profiles. Removing a duplication of effort, this reduces development time and effort. Further, the ability to reuse a previously created PIM as the base from which a number of different implementations (based on different technologies) can be derived is perhaps one of the biggest advantages of the MDA.

Interoperability benefits are one of the major proposed advantages of the MDA. As platform details within the MDA are generally hidden from the developer, interoperability problems are also. Described as an "exciting side effect" of the MDA development process by the CEO of the Object Management Group [110], it is stated that because two implementations can be derived from a common PIM (which defines a single set of data types) and because the mappings from a PIM to a given implementation technology are known and standardised, then generating a bridge between two implementations is a straight forward process [110]. This bridge would provide communication between two given implementations of a PIM, thus providing interoperability between the two implementations of a system.

Taking this concept and coupling it with the ability to reuse a given PIM to generate an implementation for whatever technology is to become popular next, it is then possible to realise a much more powerful vision of future-proofing. With the standardisation of mappings for the next technological advancement, it is possible to generate a bridge between a new implementation and any legacy implementation based on a technology that also has a set of standardised mappings (under the same bridging process proposed by the OMG). Basing the MDA around open, supported standards allows all models, data types and entities to be represented in a single, consistent manner. In this environment, both interoperability and reuse can thrive.

Improved productivity

While the OMG is "not claiming to generate all your code" [110], the emphasis of automation in the MDA process allows for clear productivity benefits. Through the use of tools that leverage automated transformations and code generation, large benefits in developer productivity are possible. In turn, this reduces the costs associated with developing an application. Further, as code generation is widely recognised as a method for increasing the quality of application code (through the removal of human error) [84], maintenance costs are also reduced, thus increasing the return on investment for a given development.

4.3.3 Shortcomings and Failures of the MDA

Within the modelling and simulation community, support for the MDA has been significant. However, while the benefits proposed by the OMG are numerous and substantial, sizeable resistance and criticism has been raised within the software development community. While the advantages of the MDA function smoothly in theory, the lack of concrete details has rendered their realisation difficult. This section discusses some of the objections and shortcomings that have been identified.

UML: The "Unwanted" Modelling Language

When the Unified Modelling Language first appeared in 1996 as a combination of other notations that existed in the software development world [76], it became the standard for describing systems in a notational form. UML is taught in universities as part of every software engineering undergraduate degree and is known to some extent by all professional software developers. However, given the goals of the MDA, is UML the appropriate option for the specification of models?

Perhaps the largest question surrounding UML is the extent to which it should be used in the development process. As mentioned above, UML is extremely popular and understood *to an extent* by every professional software developer, and therein lies the problem. Being very well suited for conveying ideas between developers, the notion of "UML as Sketch" [36] has long been employed in the software world. Under this concept, UML is used in a very loose style and only the core and most important or particularly difficult parts of a system are modelled formally [65]. In this context UML is used more to express the intent behind some facet of system design rather than to rigidly specify the structure and behaviour (as is proposed by the MDA). While it can be assumed that all software developers know the basics of UML and therefore can benefit from using it in a less formal, less rigid style such as that described above, the same can not be said for situations such as the MDA which demand an in-depth knowledge of the diagrams and rules of UML. In practice, UML is employed far more frequently and far more effectively when it is not used in an attempt to entirely specify the make up, processes and behaviours of an **entire** system.

The UML is not at all well suited to the task of fully specifying a system [36]. When the OMG first began its efforts surrounding the MDA it realised this fact and set out to remedy the situation by introducing new specifications designed to subsidise the shortfalls. Some of these extensions manifested themselves in the Action Semantics and Object Constraint specifications introduced earlier; others were to be part of an extension to the UML itself, embodied in the UML 2.0 specification. While the new specification includes many advancements designed at addressing the shortcomings of UML, it still fails to recognise the fundamental problem: people are either unable or unwilling to specify their systems to the extent and with the type of rigor demanded by the MDA.

When Standards Aren't Standards

The MDA is based around open standards for many reasons, the most compelling of which is the prevention of vendor lock-in. If all tools support the same standards then it is theoretically possible to develop and use a model regardless of the tool or its vendor. While this is a noteworthy advantage, it is reliant on the provision of standards that are comprehensive and complete. Those standards on which the MDA is currently based have thus far fallen short, leading application vendors to fill in the gaps in non-compatible ways.

For example, the XMI format specification has proven sufficiently open that in practice, the documents emitted by one tool cannot be readily imported into another. Resulting in development becoming dependent on the solution of a specific vendor, the reuse value of models and entities is reduced.

Beyond incompatible model representations, a much larger problem exists. As discussed above, in order to provide the ability to adequately model behaviour within models, the Action Semantic Language specification was produced [78]. Despite identifying the need for such a facility, the OMG neglected to describe the syntax for such a language [68]. In the absence of a standard ASL representation, vendors are free to create proprietary, competing standards, the use of which again reduces the reuse value of models and entities.

Development Process Complexity

While the concept of automation is central to the MDA development process, the realisation that 100% automation is not possible (thus requiring manual intervention and elaboration) serves only to introduce a new level of complexity to the development process.

In order to illustrate the new source of complexity, consider what must happen when a model needs to be changed. If the change is to the platform independent model, once it is completed a new PSM must be generated, followed by new implementation code. Further, without complete automation, further manual work must be competed at each step.

Problems arise when considering that the regeneration of a PSM means replacing the old one. Given that complete conversion is not possible, this also means that elaborations previously made to the PSM in earlier iterations may also be "replaced". Thus, rather then regenerating a new PSM and moving on to code, the old PSM must be merged with the new before expanding on it to cover the new additions which were made in the PIM. With this process complete, the **exact same** situation occurs when converting a PSM to implementation code. The first rule of code generation is that *under no circumstances should generated code ever be edited* for just this reason. Where previously all changes would be done at the implementation level, under the MDA there are now three levels at which a model (and the changes to that model) must be maintained. This dramatically increases complexity and as work must be done at each step, fails to insulate a developer from implementation platform considerations.

While the process becomes complicated in the situation described above, consider the impact of altering a PIM for legacy implementations. Removing an entity that is no longer required could mean the introduction of an incompatibility with all legacy implementations generated from the same PIM. These are just some of the synchronisation issues raised when considering a small change when two additional layers are added to the development process. While these are not problems that exist in the theoretical world where 100% conversion from PIM to implementation is possible, it is an example of how the MDA vision falls short.

The Myth of Interoperability

Interoperability is perhaps the most celebrated of all the professed advantages of the MDA. The ability to integrate what you have already built, with what you are building with what you will build in the future [105, 110] is a powerful vision, yet one which seems somewhat logically impossible.

The OMG states that because two implementations can be derived from a common platform independent model (defining a single set of data types) and because the mappings from a PIM to a given implementation technology are known and standardised, the process of generating a bridge between two implementations is a straightforward one [110]. However, a simple example can demonstrate the flaw in this assumption:

For a given PIM, having standard mappings to FORTRAN and standard mappings to Python-based Web Services *DOES NOT* automatically provide me the ability to build a bridge between these two platforms. In order for a bridge to be built, there must be some underlying capability within the technologies themselves to support this. While this example may seem extreme (and it is) it does demonstrate the point that unless there exists some capability in the underlying technologies, a bridging cannot occur, regardless of the existence of standardised mappings.

If a bridge cannot be built, then legacy implementations cannot be leveraged and must either be lost or re-implemented. Whatever the situation, the MDA does nothing to enable this ability to integrate or interoperate with legacy implementations, despite claims to the contrary.

Culture Concerns

Numerous technical shortcomings and questions regarding its technical viability plague the MDA. However, beyond these practical shortcomings, there is also a culture problem that envelops the MDA community.

The OMG has been a target of significant criticism for producing standards without a reference implementation. The various standards are designed by a committee made up of paying industry members, with seemingly little thought given to the practicality or useability of what they are producing. In any situation where standards are produced without reference implementations, a number of problems arise.

Technical problems that would have otherwise been recognised and rectified form part of the formal specification. While they can sometimes be minor, they can also lead to incompatibilities even *between* various sections of the specification. In the case of the MDA, which depends on a number of different specifications, the combinations of errors results in standards that gain reputations are notoriously difficult to implement and use.

Another major concern with unsubstantiated standards is that they are open to wide interpretation by the various implementers (tool vendors in this case). Without a common point of reference for resolving interpretation conflicts, different vendors can produce tools that are standard-compliant, yet incompatible with one another.

To exacerbate this problem, a number of vendors have begun to market their various offerings as "MDA compliant," without actually embodying a development process that conforms to the lofty goals of the PIM->PSM->Implementation process. At the same time, the OMG have failed to act on this situation, preferring to acknowledge these tools as proof of the MDA's viability.

With the standards body openly recognising tools that do not meet their definition of the MDA as a success, it becomes clear that vendors do not have to address the difficult sections of the process (the sections that provide the actual innovation) and can still rely on the OMG for support. As a result of this mismatch between vision and "realisation," it has become increasingly difficult to identify an MDA-compliant tool on feature set alone (rather than marketing). Without any firm criteria by which compliance is measured, the value of being "MDA-compliant" ceases to exist.

4.3.4 Summary

The MDA has been identified as a potential source of many benefits for the modelling and simulation community [85, 117]. Sadly, to this point in time, the MDA has failed to live up to its lofty expectations, as the combination of technologies chosen by the OMG to help realise the vision of the MDA is unable to meet its demands. Despite not yet achieving its goals, the motivations behind the MDA do raise a number of critical points, especially in the context of M&S.

The MDA promotes a strict separation of the **pure model** or process from the underlying technology required to implement it. As seen with other technologies presented in this chapter (component models for example), such an endeavour can help increase the reuse of that model and its entities and reduce development complexity. While the move to visual-based development techniques such as UML may not be ideal for capturing the behaviour of complex systems, the simplification of development it was designed to bring is sorely required when considering the HLA. Finally, the pursuit of automation and code generation techniques help to reduce development time, increase quality and insulate a developer from low-level details.

The need for these attributes to be some how amalgamated into the way HLA-based distributed simulations are written has been identified within the community as a requirement of the utmost importance [117]. Although the MDA fails to remedy the problems, developments such as Aspect-Oriented Programming provide an alternative that comes from similar motivations, yet is viable now – having already gained widespread use within the open source world. The next section will focus on this technology and discuss how its application can address the underlying concerns motivating this research.

4.4 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a technology whose motivation is very similar to that of the MDA: separating business logic and implementation. However, unlike the MDA, AOP works at a level that is familiar to any software developer, the application code. AOP is a methodology based around the management of *concerns*. A concern is a specific requirement or consideration that must be addressed in order to satisfy the overall goals of a system [24]. Software systems are the realisation of a combined set of concerns. For example, a banking system might involve the combination of concerns covering account management, interest calculations, statement generation, funds transfers, account information persistence, authorisation management, logging and so forth [56]. These concerns can be grouped into two distinct categories:

- **Core Concerns:** Concerns that directly address primary domain requirements and are central to the behaviour of a software system
- **Crosscutting Concerns:** System wide peripheral requirements that cut across many other concerns

In the example above, concerns such as account management, interest calculation, funds transfers and so forth would be considered *core* concerns. These application or domain specific problems are central to the behaviour of the solution. On the other hand, concerns such as persistence, authorization and logging are "system-level" problems that cut across many other concerns [24]. For example, authorization is required for account management tasks in addition to funds tasks (among others).

Object-Oriented Programming (OOP) is the most common methodology employed for the creation of software systems today. While OOP provides an excellent environment for modelling and modularising core concerns, significant issues arise when attempting to manage crosscutting problems. When implementing functionality that intersects and interacts with many concerns, OOP approaches typically result in tightly coupled solutions that are difficult to maintain and reuse. These traits are also some of the primary problems often associated with the HLA and cited as characteristics that restrict its broader uptake.

AOP represents a new methodology that builds on OOP, addressing these problems by providing a unit of modularisation known as an *Aspect* [102]. Aspects are designed to encapsulate and centralise the manner in which crosscutting concerns interact with a given application. The final system is composed by an *Aspect-Weaver*, which combines the core and crosscutting Aspects (according to a set of rules) in a process similar to code compilation[5]. This allows concerns to be developed independently of one another, removing the tight coupling otherwise required.

This section introduces and explores AOP, outlining its motivations and development processes.

4.4.1 Introduction to AOP

Identifying and abstracting software systems into sets of concerns has long been recognised as one of the best methods for reducing complexity [102]. The "separation of concerns" approach advocates breaking a system down into modules, each of which provides a well defined, related set of functionality, and can be treated as a group of opaque entities [34]. As the inner workings of such modules are hidden from external view, the complexities involved in their development and behaviour does not infect other modules that use it. Further, modules are implemented independently, and have no dependence on other modules, thus making them more portable. To provide points at this models can be pointed, contained entities can implement common, standard interfaces, allowing entire systems to be constructed from loosely coupled components, rather than a group of elements that share common interdependencies on one another [30].

OOP provides powerful semantics that easily accommodate the creation and encapsulation of functionality as modules. Its ability to support a separation-of-concerns approach and simplify software development has seen it become the standard methodology used for constructing and implementing software systems [56]. However, while capable of specifying *core* concerns, OOP does not handle *crosscutting* problems well.

By their nature, the implementation of crosscutting concerns span many separate modules. Although the OO methodology supports encapsulating and separating core problem concerns, crosscutting interests necessitates the pollution these modules with system details.

[5] Depending on the AOP framework used, the weaving process can be completed in one of two possible manners: static or dynamic. Static weaving is much like the regular code compilation process where source is compiled into some executable artefact. Dynamic weaving involves the alteration of modules as they execute and does not require an intermediate weaving step.

Consider as an example the system-level crosscutting concern of logging. As the various modules involved in an implementation complete some work, they log their activity, providing a persistent record that can be used to identify the actions taken or defects in behaviour. However, even in situations where the core logging functionality is itself modularised, invocations of that module must still be directly implanted in the implementations of other concerns. Figure 4-5 demonstrates this visually:

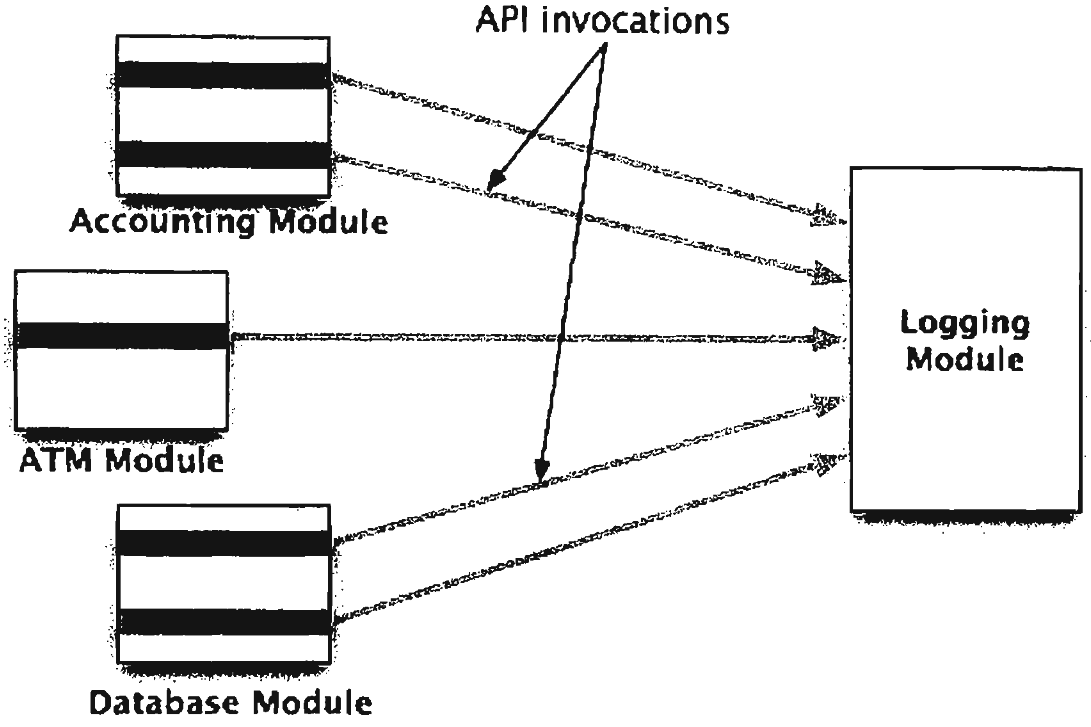


Figure 4-5: Code Tangling (reproduced from [56])

When attempting to log its actions, each implementation module must directly invoke the logging component, thus creating a dependency on it. This process is known as *code tangling* as the details of multiple concerns becomes tangled within one another. Code tangling introduces an unnecessary dependency and results in modules that are tightly coupled and which cannot be considered individual, reusable entities.

It is in crosscutting situations such as these where OOP fails to provide adequate support for proper modularisation. While the `Accounting`, `ATM` and `Database` modules encapsulate the logic for their respective core tasks, there is no support for quarantining the logging module. Some effects of this situation include [33, 92]:

- **Higher Complexity:** With implementation details of multiple concerns scattered throughout the core ones, a developer must have some knowledge of all facets involved. Further, the tangling of multiple concern details makes the implementation more difficult to comprehend and enforces a larger mental load on a developer as their mind switches between contexts

- **Poorer Quality:** Tangling makes it difficult to examine an implementation and spot problems or defects (given the constantly high level of background noise in the form of crosscutting concerns). This in turn leads to more bugs and errors
- **Increased Maintenance:** The higher complexity and lower quality of work results in increased maintenance, and as such, increased costs
- **Lower Productivity:** Because of concurrent implementation of multiple concerns, a developer's focus must constantly shift between primary and periphery considerations. Time is lost as developers are distracted from the prime objective
- **Reduced Reuse:** With modules implementing multiple concerns, other systems that require similar functionality may be unable to reuse the implementation if they have an altered set of crosscutting requirements

To address these problems, the AOP methodology defines a new unit of modularisation know as an *Aspect*. Aspects represent the implementation of a pure concern, be it business case driven or crosscutting. Where OOP would require the implementation of the crosscutting concerns to pollute that of the core ones, AOP compels developers to omit such details and implement all concerns in isolation [30]. It is important to note that AOP builds on previous technologies and as such, OOP is still used to build Aspects[6].

Once all concerns have been implemented, a set of rules defining how each Aspect maps to the others must be defined. These rules describe when the functionality of a certain Aspect should be invoked [34] in the context of the other modules. These rules they take the place of code that was previously scattered throughout the implementation of the various concerns. Thus, the code implementing each Aspect remains independent of others, with affiliations defined in separate mapping rules that can be thrown away or altered without modifying the Aspect itself.

The mapping rules, along with each of the Aspects involved, are passed to an *Aspect Weaver*. A weaver will take the defined mapping rules and use them to bind the various aspects together. For example, at the points defined in the mapping rules, code will be inserted into the core aspects to invoke logging functionality. Through a compilation-like process, the weaver will bind the Aspects together and produce the final system [23].

[6] AOP does not explicitly build on OOP. While OOP is the most commonly used approach for the implementation of Aspects, AOP frameworks do exist for older, functional programming languages. In the context of this research however, the focus will remain on OOP given the overlap that exists between it and the HLA.

A potential point of confusion often encountered with AOP is the fact that the final system exhibits all the problems that AOP claims to solve. From a logical point of view, the final implementation still looks like that presented in Figure 4-5, where the constituent modules exhibit explicit dependencies on a certain set of other modules. The key to understanding the AOP methodology is to realise that although this is true, it is only when the final system is being constructed that the interdependencies are manifested. AOP pushes the tight coupling and tangling problems away from the development context. The implementation of modules occurs separately, and each module remains an independent entity that can be reused in another application (given a different set of weaving rules). On the other hand, OOP requires that the artefacts produced by a developer display the problems identified above, and that these problems be confronted at development time.

A final AOP system, with all its flaws, is just the result of the weaving process in much that same way that the low-level details of executable binary code are just the result of the code compilation process. An application can be composed as a combination of any set of Aspects, without the need to modify their implementation directly. However, if the concerns were mixed at the source code level (as is necessary with OOP), a developer would need to make manual changes, and those changes would be forever tied to the module they were made in.

The fact that AOP is able to build on OOP approaches and leverage its advantages is a significant aid. While many of the AOP concepts are new, the process of building software applications is not fundamentally altered in any incompatible fashion. When comparing this to an approach such as the MDA, it becomes clear why AOP has already achieved significant success and been readily used and deployed in widely popular frameworks (such as the Spring Framework often used in web application development [122]). Concrete tools that conform exactly to the AOP methodology already exist and are useful in production environments.

The AOP Methodology

The development of systems using AOP is very similar to that of other methodologies. Broadly speaking, the process can be broken into three steps:

1. **Aspectual Decomposition:** In this step, the application requirements are decomposed into the set of core and system concerns required to realise them. Each required module would be identified and its interfaces designed and specified such that development can progress to the next stage.

2. **Concern Implementation:** The typical implementation phase. Modules are developed in an entirely contained and independent fashion. Object-oriented programming languages are generally used for this task, although the use of alternatives is not restricted. Once the set of modules has been implemented, the next stage can be completed.

3. **Aspectual Recomposition:** The final stage of the AOP process involves the weaving together of the various modules. Rules defining how each concern fits together are defined as *Aspects*. These rules are fed to the Aspect-Weaver that will then update the modules (creating the required links) and produce the final system. Depending on the AOP framework in use, this process can be static (compiler like) or dynamic (weaving occurs at runtime) [2].

Figure 4-6 below has been reproduced from [56] and provides a graphical overview of this process:

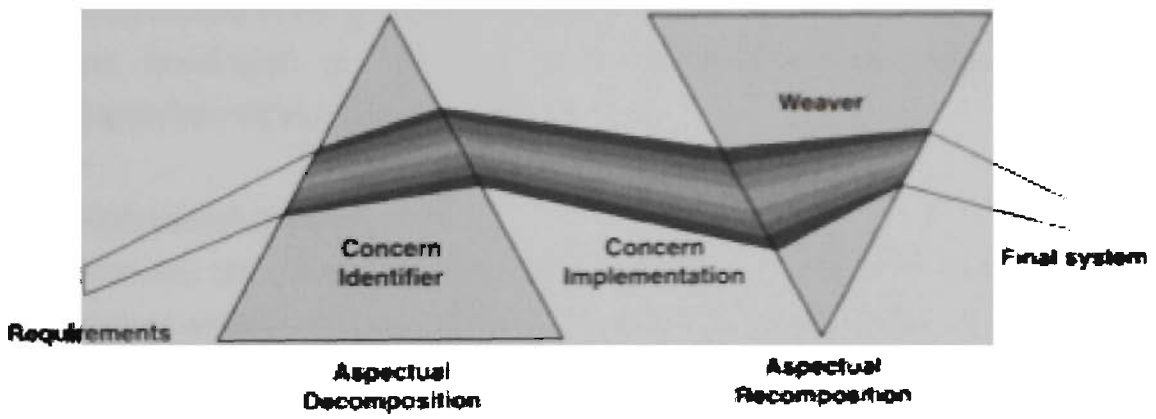


Figure 4-6: AOP Development Stages (reproduced from [56])

In the diagram above, the requirements of a system are represented much like a beam of light. During the aspectual decomposition stage, the separate concerns are identified and the application is broken down into modules. The final step involves the creation of Aspects that are passed to the weaver. The weaver then combines the individual modules together and produces a single final system.

Benefits of AOP

The process defined under the AOP methodology results in several beneficial outcomes when compared to the alternatives it builds on. These include:

- **Increased Separation**

Under the AOP methodology, modules are developed in isolation from one another. Links between disparate modules are only created by the aspect-weaver when producing the final system. As such, systems are created from a set of loosely coupled components while the persistent development artefacts are kept separate. This separation-of-concerns reduces *code scattering*, where pieces of a concern are implemented in multiple modules. Scattering can be thought of as another face of code tangling. Where tangling refers to the way separate concerns becoming entwined with one another, scattering refers to the way code for a specific concern is spread across many places.

- **Simplification of Development**
  The development of AOP applications focuses on the independent creation of individual concerns rather than the complex combination of multiple core and crosscutting considerations. Models or programs under development are easier to comprehend, as each concentrates on a single issue. The core problem does not become polluted with the lower-level system concerns required to provide crosscutting features such as application distribution or persistence. As modules are developed in isolation, problems arising from buggy or incomplete dependencies are also reduced.

- **Enhanced Reuse**
  Modules are developed separately and only combined as a complete and final system according to an independent set of replaceable rules. As such, modules are generally independent of a given application and can be easily reused in other contexts. By specifying additional weaving rules and passing a different set of modules to the Aspect weaver, entirely new systems can be created from components not originally explicitly for that particular environment. Enhanced reuse also increases the ROI for a particular development.

- **Easier System Evolution**
  AOP separates individual concerns such that they are oblivious to the components they are coupled with [56]. As such, extending a system to implement additional concerns is vastly simplified. With the creation of new mapping rules defining how a given concerns must interact with or crosscut existing modules, systems can be extended without needing to manually edit the existing implementation.

- **Reduced Development and Maintenance Costs**
  Partitioned and modular systems are simpler to compose and easier to understand. As modules are not dependent on one another, their development can occur in

parallel, saving time and reducing costs. Further, the simplified process of system evolution mentioned above helps to reduce ongoing development and maintenance costs.

4.4.2 Working with AOP

The previous subsection introduced and investigated the AOP methodology and the advantages it can provide. It has been established that AOP can enable enhanced modularisation and isolate crosscutting concerns in a way that is not previously possible. This section takes a deeper look at the processes and entities used to make AOP work.

AOP Languages

Weaving rules define *what* actions to perform *when* certain points in the execution of a program are reached. While core and crosscutting modules can themselves be defined using existing programming techniques (OOP most notably), they do no posses the requisite expressive ability to describe how crosscutting behaviour should be woven into the implementations of core concerns. As such, a special AOP language is required.

It is important to note that there is no canonical AOP language. The underlying theory behind AOP was born out of research conducted by Gregor Kiczales and his team at Xerox PARC [56]. There exist many different AOP implementations and languages, each of which is targeted at a particular platform or programming language[7]. This thesis uses one of the most widely used AOP implementations: AspectJ (for Java) [53]. As such, all the AOP examples throughout this thesis focus on the Java programming language.

Join Points

A *join point* is a location in an applications code where AOP can be used to alter behaviour. Join points are quite low level things. For example, a join point would exist anywhere a new object is created, anywhere a method is called, anywhere a variable has a value assigned to it, etc...

Join points are like passive location markers within an application. When using AOP, a programmer will attempt to "capture" a number of join points that satisfy some general pattern. If you consider an entire application as a whole, the set of join points that make up the application really just represent all the *possible* places where new behaviour or data could be woven in.

7 Implementations based on Java seem to be very popular, with many different frameworks existing, each providing different value-adding features.

Point Cuts

A *pointcut* can be considered an opening through which new behaviour is inserted into an application. Pointcuts define a set of patterns that capture a particular group of join points. They allow a user to define a number of patters and to group them under a single name. For example, consider the following pointcut definitions:

```
/** pointcut to exclude things we're not interested in */
protected pointcut ignoreList() :
      !within( hla..* ) &&
      !within( simspect..* ) &&
      !within( com.lbf..* ) &&
      !within( org..* ) &&

/** pointcut to get all consturctors */
protected pointcut constructors( Object newObject ) :
      initialization( public *.new(..) ) &&
      ignoreList() &&
      target( newObject );
```

Listing 4-1: Point Cuts

Here, two pointcuts are defined: `ignoreList` and `constructors`. Each of these pointcuts describes a number of patterns that in turn describe a set of join points to capture (or not to capture). The `ignoreList` pointcut is used as a convenience to describe all the Java packages that contain code the user is not interested in capturing. The `constructors` pointcut specifies that public constructors for <u>any</u> class (and contain any number of arguments) should be captured, as long as the classes do not reside within any of the packages specified by the `ignoreList` pointcut.

Advice

In AOP, *Advice* is the name given to the logic that is inserted into an application through all the join points identified by a point cut. It is the advice that provides the new code that gets woven into the model.

When defining an advice, a user must also declare how that advice is to be woven into a model relative to existing code. Given a pointcut that captures all calls to the `connectToDatabase()` method, should the advice be inserted before the call, or after the call? The following example demonstrates a different kind of advice known as an "around" advice:

```
/** pointcut to get the main method */
protected pointcut mainMethod() :
      execution( public static void main(String[]) );

/** advice to decide whether or not to execute main method */
void around() : mainMethod()
{
      if( Math.random() > 0.5 )
            proceed();
      else
            System.out.println( "Not running main method" );
}
```

Listing 4-2[8]: Advice

Here, the around advice wraps around the execution of any method that was captured by the mainMethod pointcut. Unlike other kinds of advice (such as before or after advice), around advice can determine whether or not the method that has been captured can proceed. In this case, if the result of a random number generation is greater than 0.5, then the method call is allowed to proceed.

Through these facilities, points in an application can be identified, and new logic or data can be woven in. This is what provides the AOP development model with its power. If a developer was attempting to weave together their business modules with a system logging Aspect, the code that would call into the logger (the advice) could be inserted through a point cut at all the join points that exist for the beginning of a new method call. This would allow the core logic to remain small and clean.

4.4.3 AOP Viability

Sharing similar motivations, the MDA and AOP have much in common. However, AOP currently maintains one crucial advantage: it works now. From the standpoint of functionality, the MDA is still a long way from being able to achieve its goals. On the other hand, there already exist many mature AOP implementations. Within the Java community, AOP widely used either explicitly [57], or as part of a larger framework (such as the immensely popular Spring framework) [13, 60]. These implementations have already gained wide acceptance and are being used to power many mission critical applications. Beyond these examples, many AOP implementations exist for many different

[8] For those who wish to see a full example of how Advice is specified, the full code for the reference implementation developed in this thesis is provided in the supplementary package that accompanies this thesis. Alternatively, [56] provides many excellent examples.

programming languages, with [127] listing well over 50 such projects for more than 15 different languages.

4.4.4 AOP: A Potential Solution?

Significant overlap exists between the issues that motivate AOP and those that currently afflict the HLA. Many of the current development shortcomings stem from the tangling of pure model code with a complex distribution technology. This in turn obscures the real value of a simulation (the model itself), while tight integration and coupling reduce any reuse potential and return on investment. The AOP development process has the potential to considerably ease these limitations and help enable a broader uptake of distributed simulation.

The HLA is a system-level crosscutting concern, the focus of which is the distribution of simulation information between many disparate simulation models. Distribution technologies like these typically suffer from the tangling of concerns discussed previously, making them an ideal candidate for AOP [21]. However, although AOP methodology prescribes a separation of concerns, such an approach would still necessitate the authoring of a HLA Aspect and weaving rules that define where it crosscuts the core concerns. The development of HLA Aspects requires skills and training that do not exist within the wider business community. Further, depending on the needs of a simulation with regard to elements such as time and execution management, the requirements for such an Aspect could vary widely.

To address this problem, the development of a Generic Aspect is needed. Consider Figure 4-7:

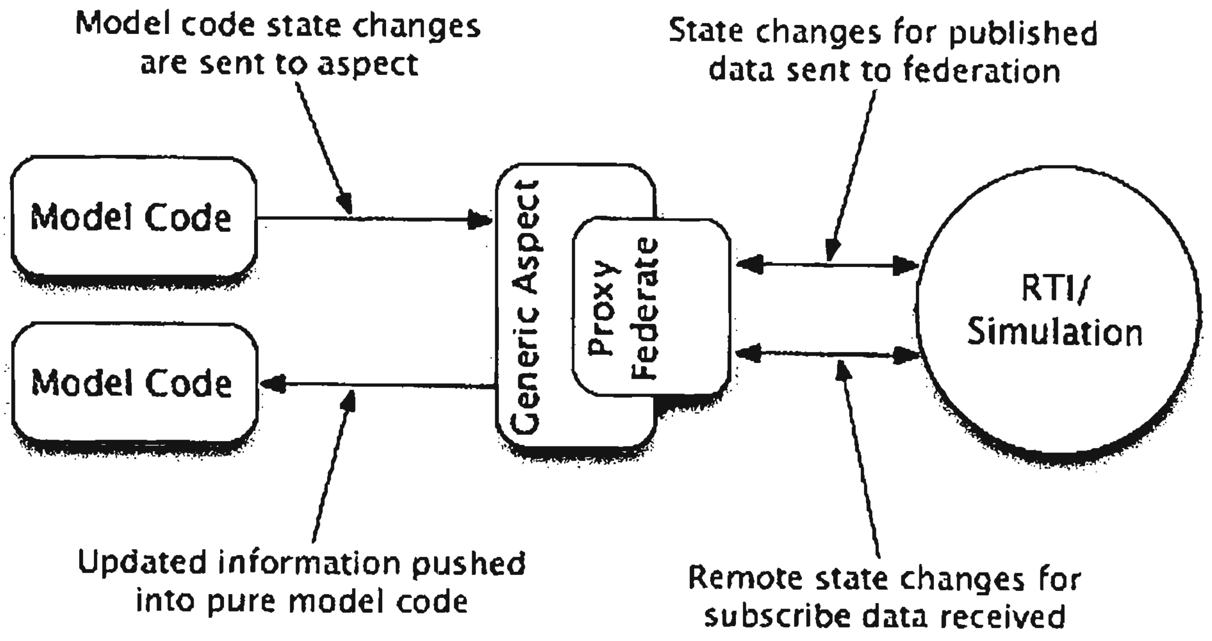

Model code state changes are sent to aspect
State changes for published data sent to federation
Model Code
Model Code
Generic Aspect
Proxy Federate
RTI/ Simulation
Updated information pushed into pure model code
Remote state changes for subscribe data received

Figure 4-7: A Generic Aspect [CE]

In this figure, changes within the model trigger logic within a Generic Aspect. Through the process of Aspect weaving (via a general set of rules), the Generic Aspect can capture changes within the pure model code. As changes occur in the model, they are pulled into the Generic Aspect and processed by the proxy federate. All interaction with the HLA is isolated from the pure model code. Changes made in the simulation are pushed back out of the proxy, into the pure model.

Generic proxy federates have been used to achieve similar ends in other applications. The goal of the fedWS2 project was to provide access to information from an active simulation in a manner that hides the underlying HLA details [88]. A configurable generic proxy federate was successfully used to allow interaction with any HLA simulation without the need for end users to author code [89]. The use of a proxy environment also provides an ideal location into which additional supporting technologies (such as those described in section 4.2) can be deployed. To help solve common HLA problems such as FOM Agility, approaches defined in existing research can be deployed within the proxy, alleviating those issues identified in chapter 4.

Leveraging a generic HLA Aspect in combination with AOP methodology yields a solution that would allow **pure model code** to be developed independently from any HLA concerns. Figure 4-8 presents an overview of how this process would occur:

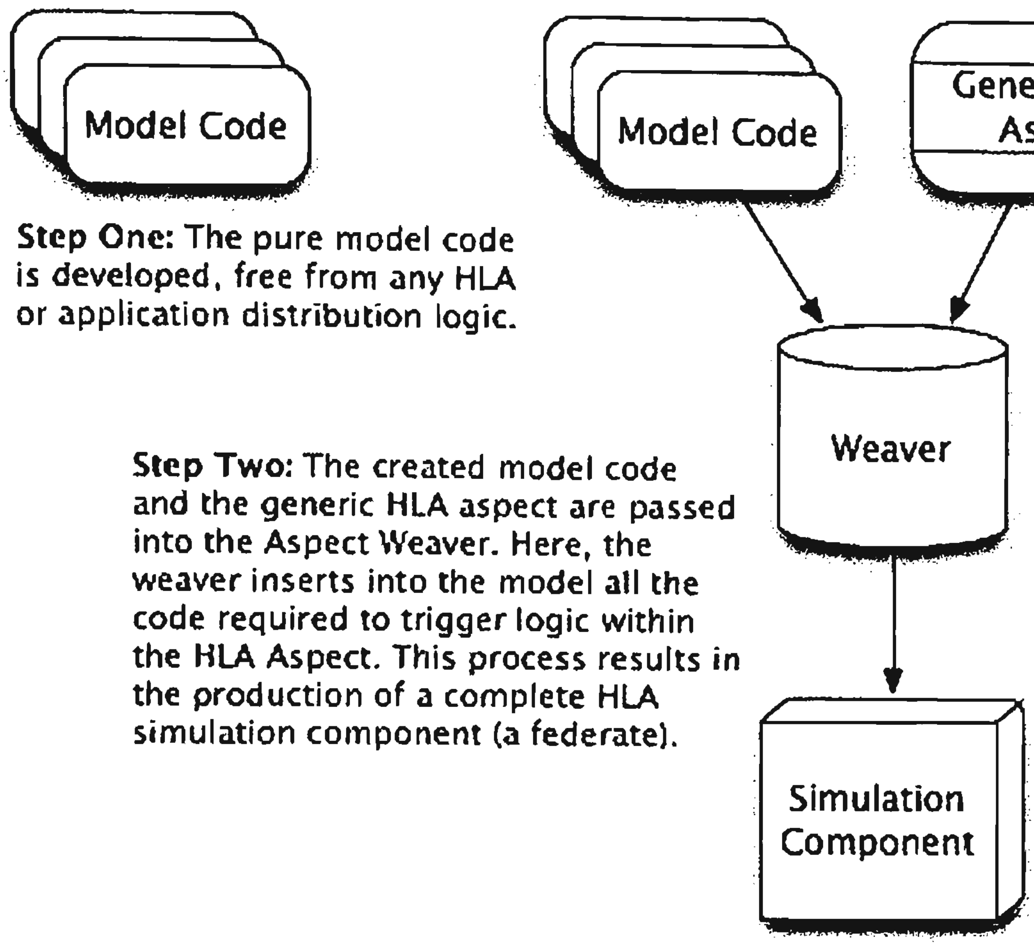

Model Code
Step One: The pure model code is developed, free from any HLA or application distribution logic.
Model Code
Generic HLA Aspect
Weaver
Step Two: The created model code and the generic HLA aspect are passed into the Aspect Weaver. Here, the weaver inserts into the model all the code required to trigger logic within the HLA Aspect. This process results in the production of a complete HLA simulation component (a federate).
Simulation Component

Figure 4-8: Simulation Component Generation Process

First, the pure model logic is developed *free from any HLA considerations*. Following this, the developed modules and the Generic Aspect (containing mapping rules) are passed to the weaver that combines them, producing the final system. Such a solution fulfils the goals of an implicit HLA simulation environment. Some of the primary benefits include:

- **Barrier to Entry:** Remove the barrier to entry that requires expert skills and training, thus alleviating one of the primary restrictions limiting a broader uptake of the HLA
- **Generally Applicable:** The AOP methodology is not tool-specific, and as such, the methods developed and discussed later in this document could be employed in any environment
- **Leverages Existing Advancements:** The use of a proxy environment provides the ideal point at which research discussed in section 4.2 can be applied
- **Reduces Development Time and Complexity:** The separation-of-concerns approach allows developers to focus on the core problem free from low-level system details

- **Increased Reuse:** As the model components are no longer tied directly to the HLA or a specific FOM, they are more readily reusable. The employment of FOM Agility techniques within the proxy environment also aids this cause
- **Beneficial to Existing Community:** The advancements made possible in the described environment are beneficial not only in enabling a broader uptake of distributed simulation technology, but also of benefit to existing HLA users

The application of AOP within the modelling and simulation environment has the potential to provide many benefits. In [112], Straßburger identifies the implicit method as allowing simulation developers to continue working in a comfortable and familiar environment, while benefiting from the advantages use of the HLA enables. The use of AOP in the manner presented here can help realise a more generally applicable version of the implicit approach. While Straßburger focused on the development of tool-specific interfaces in the pursuit of realising his goal, this research seeks a broader, more generally applicable solution.

AOP techniques are not tool-specific, and as such could be applied in many environments. Many AOP implementations for many programming languages and platforms exist, meaning that likelihood that AOP could be incorporated into a given simulation tool is high. Through the use of AOP, the methods developed within this research can safely ignore the technical nuances of any given simulation tool, instead focusing on generic techniques that are applicable in a broad sense. In situations where simulation code is to be developed directly, AOP provides many immediate benefits with regard to the separation of core and crosscutting concerns. Where models are constructed and executed with specialised, proprietary tools, AOP platforms can be incorporated into those tools, also enabling them to realise the same benefits.

4.5 AOP Shortcomings

Section 4.4 introduced AOP as the most viable source of potential advantages with regard to the goals of this research. However, use of AOP alone cannot provide a complete or comprehensive solution to the problems motivating this work. Of primary concern is that AOP only describes a methodology for separating the development of core, business logic modules from that of the crosscutting system modules. There are a number of questions beyond the domain of AOP that must first be addressed before the goal of automatically rendering pure models as HLA simulation components can be reached.

Aspect-Oriented Programming mandates that the development of different system concerns be completed separately. This way, the individual modules that are created do not form any dependencies on system level tools (such as those used for authentication, persistence or application distribution). Additionally, this process simplifies system development, as the needs of multiple modules, spanning both core and platform logic, do not need to be considered. It is only at the final stage, when the system is being composed from the collection of modules, that the pure business logic is coupled with the required system-level details.

This type of development process fits very well with the objective of simplifying HLA simulation development. Pure model modules can be constructed independently of the HLA, and be bound together at the last minute, rather than forcing HLA concerns to be handled throughout the entire development process. However, from the perspective of this research, a number of problems remain.

The HLA Aspect

Use of the AOP process is predicated on the development and availability of a HLA Aspect. Although model code can be developed independently, unless there is a HLA Aspect to weave into it, a HLA-compliant component will never be produced. However, development of such an Aspect would necessitate intimate HLA knowledge, a requirement that has previously been identified as insufficient in the context of this research.

Weaving

Although AOP facilitates a separation of concerns approach during development, the process of defining how modules are to be woven together still necessitates a fundamental understanding of both the core and crosscutting concerns. Even if a HLA Aspect existed, decisions regarding where HLA behaviour must be inserted into the pure model must still be made. Further, users must be able to identify the *type* of HLA logic to weave in at these points. Once again, this demands specialist knowledge of the HLA that is not tolerable within the goals of this research.

Deployment Artefacts

A central artefact in any HLA simulation is a shared model (known as the FOM), defining the vocabulary of information exchanged between the participants. This is required to perform actions such as publication and subscription, or to register and update attribute values. Without this information, an individual simulation cannot function correctly in the shared space, and as such, is not of any use. Although AOP provides support for intercepting actions within a model and injecting specific behaviour at those points, it provides no facilities for generating the necessary artefacts or configuration information required to deploy a federate. The lack of support for these types of artefacts is to be expected as they often fall beyond the realm of source code (the level at which AOP operates). Additionally, this problem is HLA specific, and only of concern in a HLA environment.

Federate Level Agreements

Each HLA-based distributed simulation involves the co-operation of many individual simulation components. While the interesting portion of a simulation is the data it produces during execution, there are many additional "housekeeping" actions that must be performed.

Each federation has its own (potentially unique) series of steps it takes to ensure that components are able to synchronise with one another and work in step. There are understandings in place that define how and when data will be registered, the frequency with which it will be updated and so forth. There may be yet more requirements that define how specific additional information is formatted and passed between federates. Within the HLA specification, the developers of each federation are free to define the steps they expect federates to take, and facilities they expect federates to use, in order to co-operate correctly with the other components in a federation. These facets of simulation execution are known as *federate level agreements* (also often referred to as "federation level agreements") as they define various behaviours expected at a federate level[9].

However, federate level agreements are governed much like the colloquial "gentleman's agreement," in that they are not explicitly documented (in a configuration sense) nor enforced. Failure to meet one of these agreements often manifests itself in unusual ways. While data formatting omissions may trigger explicit errors, execution management mistakes are often more subtle.

Each federate must be manually programmed to observe any federate level agreements, and this step must be completed for every federation in which that federate intends to be deployed. Given the arbitrary nature of such concerns, providing generic support for such requirements presents a somewhat insurmountable problem. Beyond the hand development of a specific agreement aware HLA Aspect, there is nothing within the AOP realm that can help address this issue.

---

[9] These are also often referred to as the Execution Management requirements for a federation.

Monolithic v. Distributed Environment Mismatch

Perhaps the biggest concern involved in rendering a pure model as a HLA distributed simulation component is the inherent mismatch between the expectations of each approach. HLA support is a system level, crosscutting concern, and an ideal candidate for being separated from business logic. However, this separation also removes considerations of *application distribution* from the development of code modelling components. Practically speaking, these models become monolithic, self-reliant implementations. In a distributed simulation, the creation and manipulation of data is the shared responsibility of all participating components. In a monolithic simulation, the model code is given unquestioned dominion over its data.

The tension between monolithic and distributed environments manifests itself in two primary ways: the problems of *data introduction*, and management of the strict *ownership* rules imposed by the HLA. In a monolithic simulation, the model is entirely responsible for the creation and storage of all data. In a distributed situation, data may be created externally. Given this, there are considerable questions about how data can be introduced into a model that is not expecting it.

Further, the HLA specification mandates rather strict data ownership rules, prescribing that only a single federate may own a piece of data, and only that federate is entitled to modify it. In the selfish realm of a monolithic simulation, the model is able to alter any piece of data at any time; it has no notion of sharing or ownership.

The HLA also introduces publish and subscribe facilities. These mechanisms are used to define what remote data a federate is interested in, and what data it produces for remote consumption. Often times this information can also be used to describe which data from a model is meant to be kept private, and only used within a federate, and which is meant to be available to other federates within a shared federation. Although object-orientation includes something similar in the form of its data access rules, these alone are not expressive enough in a distributed context. For example, consider an OO component that describes some information as having public access, thus allowing other components within the model to access it. Just because the information is meant to be available to other model entities, does not meant it is meant to be available in a distributed context. In a HLA setting, the information may still need to be marked as *public* (so that other parts of the federate can access it), yet it may not necessarily be intended for publication to the federation. Once again, this problem stems from the disconnect that exists between a monolithic and distributed environment.

Although AOP does provide a number of enticing opportunities for the simplification of HLA simulation development, when considering its use within a generalised, automated environment, there are still many questions that must be answered. As this section has shown, the remaining problems broadly pertain to the mismatch that exists between a pure-OO environment, and the distributed, shared simulation environment of the HLA. The next section builds on the information presented here and outlines a number of research questions that this research addresses.

4.6 Summary

Much of the research presented in this chapter has focused on the simplification of HLA simulation development. Through the abstraction of low-level tasks, considerable productivity gains can be made. An increased focus on separating the pure simulation model from the underlying implementation infrastructure has served to enable greater levels of interoperability and reuse. As too have techniques to help manage object model differences.

With the work presented in [112], Straßburger introduces the notations of explicit and implicit access to the HLA through simulation tools specific interfaces. While explicit access to the HLA is not suitable in the context of this research (as it also requires specialist knowledge), the implicit approach realises the desired goal. Straßburger was successful in exposing civilian simulation applications to the HLA via tool-specific interfaces. In the case of the implicit approach, this involved the translation of simulation events into the relevant HLA counterparts.

The focus of this study is to build on Straßburger's implicit approach and investigate methods for implicit model development that are generic in nature. While the solutions presented by Straßburger in [112] show how this approach can work for a particular tool, this research seeks the development of a *general-purpose* solution that can provide HLA simulation services transparently to generic simulation models, free from ties to any individual application.

AOP and the MDA both describe approaches that reinforce the idea of separating core business logic from that of implementation concerns. Although the simplicity of the MDA process is desirable, particularly in regard to its considerable use of automation, the reality is that the technology does not yet exist to back up its claims. As such, it is not suitable for use within this research. Despite working on a much lower level (one for which expertise already widely exists within all corners of the information technology industry), AOP is not only feasible, but already proven and deployed in copious amounts of production environments.

Aspect-Oriented Programming outlines methods for transparently combining platform specific considerations with those of a core problem. Additionally, the concepts and theory of AOP are not tied to any specific tool, with implementations existing for many platforms and programming languages. As this chapter has shown, AOP can form a solid foundation upon which this research can build techniques for the transparent rendering of pure models as HLA simulation components. In such a context, AOP acts largely as a facilitator, providing the ability to intercept and alter a model, allowing HLA behaviour to be injected transparently. However, the use of AOP alone does not address a number of considerable theoretical issues about the form that behaviour should take, and where it needs to be interested.

While sharing many similarities, Object-Oriented theory and semantics do not align perfectly with those of the HLA. The normal process of converting a model to be HLA-compliant necessitates human intervention. This in turn demands specialist HLA knowledge that does not exist within the wider business community. As has been highlighted time and time again, within the context of this research such knowledge cannot be expected and is deemed unacceptable. Given these constraints, the involvement of automation in the conversion process is necessary. A number of questions still require answers when considering the application of AOP to the HLA as a potential solution. As the next chapter explains, the contribution of this research is to address these problems, defining the requirements and methodology involved in automatically rendering a pure OO model as a HLA simulation component.

Chapter 5

Research Questions and Experimental Framework

This chapter outlines the the major remaining questions that blind application of AOP to the distributed simulation domain alone cannot answer. It is the answers to these questions that form the contribution of this work. The final section of this chapter establishes the experimental framework that will be used to assess and validate the solutions that are posed in the coming chapters.

5.1 Research Questions

AOP alone is not sufficient for realising the goals of this research. While providing a strong supporting framework, the use of AOP as a mechanism for entirely abstracting the complexities of the HLA raises many sizeable questions [92]. This section introduces those questions that drive this research.

Object Models

Shared object models are a central part of the HLA. While the notion of an object-hierarchy is implicit to object-oriented programming, a HLA object model requires more than pure lineage and inheritance information. Without the requisite manual direction, some method for automatically extracting object model data must be devised.

"How can a HLA object model be extracted from a pure OO simulation model?"

Public and Private Data

Through join points and point cuts, AOP allows for the specification and insertion of crosscutting behaviour. In a typical AOP situation, a developer would use semantic understanding of the model to identify where certain point cuts would need to be made and what advice would need to be inserted at those locations. However, linking a model and the HLA Aspect would require specialist knowledge. Without semantic understanding, determining data that is public and meant to be shared with a simulation from that which is private, and meant for internal processing, becomes difficult.

"How can the public and private data of a pure model be automatically identified?"

and

"How can the publication and subscription requirements of a pure model be identified?"

Object Data

The concept of an "object" is a common link between OO and the HLA. Although the notion of an *object class* carries different meaning in both, the notion of an object instance representing a unique set of data is shared idea. However, for a distributed simulation to function correctly, data must be shared between the participants. How this behaviour can be transcribed into a model that has no distribution concerns presents a problem.

"How can the creation, removal and alteration of data within an OO model be replicated into an active HLA federation?"

Many of the problems this research addresses are inherently linked. This question is intimately connected to the problems of public/private data identification, as only changes to public data should be shared with a federation. Further, this question is also linked to the following problem relating to data introduction.

External Data Introduction

AOP provides excellent facilities for capturing changes made to pieces of data within a model. At these times, if the information is relevant to the greater simulation, it can be easily sent to the federation. However, problems begin to arise when pondering how data created and managed in remote federates can be feed into an object-oriented model that would not be expecting it. Although central to the HLA, OO models have no notion of application distribution; and as such, the introduction of foreign information is a significant concern.

"How can the creation, removal and alteration of data within an active HLA federation be replicated within a pure OO model that is not expecting it?"

As pure simulation models are monolithic in nature, they have complete control over their data and implicit permission to alter it at any time. Under the HLA, the rules of data ownership contradict this and will not allow for such events to occur.

"How can the data ownership rules of the HLA be reconciled with the monolithic world-view of pure object-oriented models?"

Grouping Behaviour: Methods and Interactions

It is generally accepted within the HLA community that the relationship that exists between interactions and OO-style methods is parenthetical. Although similar, they were each designed with distinct purposes in mind. In object-oriented methodology, methods are directly associated with a specific object type, and are considered to describe the behaviour that the type may implement. Interactions on the other hand are designed more like messages. Although their structure is independent of any object class, they may be arranged within a hierarchy with regard to one another. This is a significant point of difference.

Regardless of these differences, the fact remains that methods play a vital role in the development and execution of object-oriented application, and as such, the actions they perform (and the consequences of those actions) must somehow be translated to the HLA world.

"How do object-oriented methods translate into HLA interactions?"

Federate Level Agreements

As highlighted in section 4.5, with any HLA federation there are a number of undocumented agreements that describe how federates should behave in certain situations. These federate level agreements are generally the domain of execution management concerns. Although they are not always consequential to the actual core processing of a simulation, they do express the requirements necessary to cooperate with other components.

These agreements can generally be considered housekeeping requirements for executing a distributed simulation, and as such, are specific to the HLA. In the monolithic realm of a pure-OO model, there is no need for such considerations, and as such, there is no parallel from which an automated process can extract the necessary information. This raises a further concern:

"Can the definition of federate level agreements be expressed without requiring manual intervention?"

Logical Time

Any given simulation model may represent logical time in any number of ways. Other models may choose to ignore the concept of time entirely. As it currently stands, many within the HLA community consider time services to be a periphery consideration.

However, when time is used, one of the primary advantages of the HLA is that it provides facilities for the shared management of advancement. Federates are able to keep synchronized with one another through adherence to a single shared value, whose advancement is controlled by the RTI (as discussed in section 3.2.3.4).

Given the multitude of different ways in which logical time can be represented in a model, it becomes virtually impossible to automatically pick how a given model handles this facet. However, if this information is known, is there even a way to enforce synchronization between a pure model and a federation? As highlighted above, pure models are monolithic in approach, and expect the freedom to modify any variable at any time (including that which may represent time). In the shared environment of a distributed simulation, these actions need to be controlled.

"How can logical time be synchronized between a monolithic pure model and a shared distributed simulation?"

Authoring Distributed Models

The questions introduced above share a common thread: they stem from the inherent misalignments that exist between the object-oriented and HLA worlds. While each question is driven by a significant problem that arises when attempting to combine these two distinct ontology's, they all form part of a larger, deeper question.

"How can pure models, that know nothing of application distribution, be created to depend on and work co-operatively with other remote models?"

Part of the power of distributed simulation is that it allows work to be partitioned among a number of disparate models. However, with no notion of application distribution, each pure model is essentially monolithic. This raises questions about how they can be designed to participate in a co-operative environment, where some of the information will be generated and manipulated remotely.

Other HLA Services

The questions presented in this section outline the areas of OO/HLA crossover that this research considers. While the list is comprehensive, not all HLA services are addressed in-depth, or even at all. Some services, such as time management, could easily form the basis of entire theses in their own right. Addressing these periphery issues in such depth is well beyond the scope of this work. Discussion of other HLA services, such as Data Distribution Management (DDM) and Save/Restore support have been omitted entirely. Again, as the core focus is to address the initial problems involved in producing a generic, implicit development environment, these additional topics are of a periphery concern. Perhaps, if time possessed some infinite quality this would be possible. However, in this realm, these issues are out of scope and form a fertile target for further work (as discussed in the final chapter).

To fulfil the goals of a generally applicable, implicit HLA simulation environment, this research addresses the questions presented here. In doing so, this work forms a significant contribution to the current state of the art; vastly reducing the complexity involved in authoring distributed simulations and substantially reducing the primary barriers limiting an uptake of the HLA within the wider business community.

5.2 Experimental Framework

The goal of this research is to allow pure object-oriented models to be transformed into HLA simulation components through a process that removes the need for specialist HLA knowledge. The crosscutting nature of the HLA results in the pollution of pure model code with complex, low-level infrastructure details. Aspect-Oriented Programming represents a potential solution to these problems by separating the development of model and HLA Aspects. However, as discussed in this chapter, despite providing separation, use of AOP still leaves many unresolved questions.

To address these concerns, the following chapters propose a design for a generic HLA Aspect and a methodology for extracting HLA semantics from a pure-OO model. In combination with AOP, these two advancements can be used to automatically render pure model code as a HLA simulation component. This section describes the experimental framework used to test the validity of the solutions presented in later chapters.

5.2.1 Overview

The experimentation process employed by this research is broken down into three separate stages. Beginning with a fully manual AOP process, the experiments show how AOP can be used to quarantine HLA concerns from model development, and then how automation can be used to remove the requirement of HLA knowledge.

The main experiments use a set of two synthetic simulations that have been created for this research. Introduced below, these simulations exist only as standalone object-oriented applications (no HLA versions have been created). They have been purposefully designed to incorporate many facets of object orientation and are able to function entirely on their own. The final experiment ignores these test cases and instead puts the developed theories to test with an existing distributed simulation. This purpose of this test is to demonstrate to co-operation of HLA and non-HLA models in an existing setting.

5.2.2 The Test Simulations

This section briefly introduces each of the test simulations used in the various experiments. The code for all simulations is provided in the supplementary package that accompanies this thesis.

The Race Car Simulation

The "Race Car Simulation" was designed to model a very simplistic car race. The primary design goals of the race simulation were:

- **Repeatability:** The race should be deterministic to help ensure that any observed behaviour should occur consistently.
- **Structural Simplicity:** The overall structure (object composition, inheritance hierarchies, etc...) should remain as simple as possible. This model is meant to test the basics.

Each car in a race has a specific top speed, which in this case acts as its constant speed. As logical time advances for a race, the car is assumed to be travelling at its maximum speed. As such, the results of the race (and the position of a given car at a given point in time) are deterministic.

The purpose of this test case is to provide a baseline of behaviour, just enough to ensure that the developed solutions work, without requiring additional complex behaviour. As such, the structure of the model is kept simple, with no inheritance hierarchy and minimal object composition. For a detailed explanation, see Appendix A.

The Sushi Boat Simulation

The "Sushi Boat Simulation" is designed to be slightly larger and more complex than its counterpart. This simulation is a *loose* adaptation of the case study used in [26]. It has been designed to include:

- **Repeatability:** As with the first test scenario, this simulation has also been designed to be repeatable.

- **Inheritance Hierarchies:** The structure of the dishes forms an inheritance hierarchy that must be accounted for in both the generation of an HLA object model and the encoding and decoding of simulation information.
- **Object Composition:** This scenario includes many objects that reference other objects. While common in object-oriented environments, this type of relationship is foreign to HLA models.
- **Method Dependence:** It is common for OO models to depend on methods to carry out large amounts of functionality. This can be a problem for the HLA, given a misalignment between the concept of methods and interactions. This simulation depends on the use of methods to perform the main behaviour.

The scenario revolves around a simulation of a Sushi restaurant. There are a number of `Dishes` on offer within the restaurant, each of which is a `Starter`, a `Main` meal or a `Desert`. The meal objects form an inheritance hierarchy. Dishes are prepared and travel along a small "river" past a number of tables. At any point, a table may pick up a dish and eat it (thus purchasing it). At the conclusion of a meal, the information about all the consumed dishes for a table is calculated into a `Receipt` and the table is cleared.

For a complete explanation of this test model, see Appendix B.

<u>The Air Transport Operations Simulation</u>

The Air Transport Operations (ATO) simulation is a simulation that is primarily used as a teaching aid at the University of Ballarat [119]. The scenario consists of three main federates (although a fourth optional one is sometimes also used).

The **Aircraft Manager** (ACM) federate is responsible for creating Aircraft objects and updating their state as they fly around the simulated environment. The ACM federate receives interactions when other federates need to control its actions. For example, when an aircraft is permitted to land at an Airport, a "Land" interaction is sent.

The **Air Traffic Control** (ATC) federate is responsible for all the airports and associated Runways. It controls which planes can land at the various airports and when, potentially telling aircraft to loiter, divert or land.

The **Flight Manager** (FM) federate is responsible for deciding where each plane should fly to, how long it has to wait between flights and when maintenance is required. When a plane has landed, the FM issues it directions as to what to do next.

In the final experiment, a pure-OO Airport model will be created and placed into the existing simulation. Aircraft should then be able to fly to and from this destination as if it were created and managed by a normal HLA-federate. For this experiment, an implementation of the ATO federation developed by UoB students will be used. This implementation includes an additional federate that provides a visualisation of the federations activities.

5.2.3 Experiments

The experimentation for this research focuses on three separate stages. Each stage is necessitates the answering of a specific set of research questions, with subsequent experiments building on their predecessors.

Broadly speaking, the problem of extracting HLA specific information from a pure model and allowing it to participate in a HLA federation involves two considerations:

- Information introduced, updated and removed ***within a pure model*** must be ***distributed via the HLA***

- Information introduced, updated and removed by ***external federates*** must be made available ***within a pure model***

The provision of an environment and methodology that is capable of meeting these broad objectives is the focus of experimentation.

Each of the experimentation descriptions below is broken down into four areas:

- The **Purpose** section provides an overview of the experiment and states it goals
- The **Prerequisites** section describes the work that must be completed before the experiment can run
- The **Procedure** section defines the steps involved in executing the experiment, and how the results will be collected
- The **Qualification of Success** section defines what must be observed for the experiment to be considered a success

In the remainder of this chapter, the pure object-oriented model will be referred to as the *pure model*. The HLA-compliant version of this model will be referred to as the *AOP-model*.

The captured log file information will be analysed to determine the actions that occurred throughout the experiment.

Qualification of Success: For this experiment to be considered a success, a number of criteria must be met:

Success Criteria	Validation Method
Pure-model must remain free from any HLA considerations	Visual inspection of pure-model code.
AOP-model must execute without error in federation with companion federate	Non-Error execution of federation deemed a success.
Object-data <u>created</u> by AOP-model must be sent to federation *(objects)*	Inspection of log file for companion federate to validate objects were created.
Object-data <u>created</u> by companion federate must be received by AOP-model *(objects)*	Inspection of log file for AOP-model for presence of remotely created data.
Object-data <u>changes</u> inside AOP-model must be sent to federation *(attributes)*	Inspection of log file for companion federate to validate reflections received.
Object-data <u>changes</u> inside companion federate must be received by AOP-model *(attributes)*	Inspection of log file for AOP-model for presence of remotely altered data.
Relevant method calls within AOP-model must be sent to federation *(interactions)*	Inspection of log file for companion federate. Ensure interactions are received.
Relevant interactions sent by companion federate must be converted in method calls in AOP-model *(interactions)*	Inspection of log file for AOP-model and inspection of pure-model results to ensure methods are called.
The results generated by the pure model in standalone form ***must not*** match those generated when run with companion federate.	Compare results log generated by pure-model when executed standalone and when executed with the companion federate.

Table 5-1: Experiment One Success Requirements

An important point to note is the specification of final criteria. In its pure form, the model will produce a set of results (a list of positions for the Race simulation, or a list of receipts for the Sushi Boat simulation). When running as a HLA federate, this same model should be acting on **additional** data created by the companion federate. Accordingly, the results ***should not*** be the same as when the models are run by themselves.

Experiment Two: Consuming HLA Information

Purpose: The purpose of this experiment is to build on its predecessor and validate the automated production of all components requiring HLA knowledge in experiment one. The results yielded by this experiment should match those that were produced in the previous experiment, however, unlike its predecessor; the process used to produce those results should be free from any HLA considerations.

The first experiment requires a number of artefacts to be hand generated by a user before the AOP-based framework could be used successfully with an OO model. Before this experiment can be completed, approaches for automating the production of that information are necessary.

Prerequisites: Before this experiment can take place, a number of research questions must be addressed. These questions generally relate to how the many of the manual processes of experiment one can be automated. As we are building on the previous experiment, its prerequisites are implicit in this list.

- Automatic determination of the relevant locations within a pure model to capture information creation, modification and removal
- Automatic extraction of HLA Object Model based on the structure of the pure OO model
- Automatic determination of public/private data within a pure model
- Automatic determination of public/private methods within a pure model
- Automatic determination of publication and subscription interests for a pure model

Each of these requirements is necessary to generate all the appropriate information needed by the AOP-based runtime when interacting with a pure-OO model and to ensure that information flows smoothly between the OO model and the companion federate.

Procedure: For each of the two primary test models, the following steps will be taken:

1) The pure model will be first run standalone. The results of the simulation will be recorded
2) The pure model will be run through an **automated process**, producing the AOP-model necessary for simulation
3) The AOP-model will be run in a federation with the companion federate
4) Log file information will be captured for both the AOP-model and companion federate

The captured log file information will be analysed to determine the actions that occurred throughout the experiment.

Experiment One: Manual AOP

Purpose: The purpose of the first experiment is to validate the use of AOP as a means of abstracting HLA concerns. It will ensure that the business logic of a simulation can be developed free from HLA considerations. At the completion of this, the model and HLA Aspects will be manually mapped together, mimicking how the AOP development process would occur if HLA expertise were available. In discussions below, the term "**AOP-model**" refers to the version of the pure-OO model that has had Simspect woven into it.

Prerequisites: Before any testing simulation can be run, there are a number of prerequisites that must first be addressed. These include:

- A generic HLA Aspect must be created
- Manual determination of appropriate Aspect-Weaving locations
- Manual object model creation
- Manual determination of publication and subscription interests
- Manual determination of public and private data
- Manual handling of any execution management or federate level agreements
- Manual translation of data between OO-model and HLA

Additionally, to validate that information is indeed being sent to and received from the HLA, a custom logging federate must be created. This federate will have two purposes. Firstly, it will log the information it receives (ensuring that the AOP-model is generating the appropriate events). Secondly, it will create and modify information, to help ensure that the AOP-model is receiving remote information that is being noticed by the pure-OO code. Versions of this federate exist for both the race and sushi simulations. From this point on, this will be referred to as the *companion federate.*

Procedure: For each of the two primary test models, the following steps will be taken:

1) The pure model will be first run standalone. The results of the simulation will be recorded
2) Each of the necessary manual processes will be completed, producing an HLA-compliant version of the model, known as the AOP-model
3) The AOP-model will be run in a federation with the companion federate
4) Log file information will be captured for both the AOP-model and companion federate

Qualification of Success: For this experiment to be considered a success, a number of criteria must be met:

Success Criteria	Validation Method
All necessary artefacts must be automatically generated	Successful if process consumes pure-model and generates AOP-model that is run without intervention.
Pure-model must remain free from any HLA considerations	Visual inspection of pure-model code.
The results generated by the pure model in standalone form ***must not*** match those generated when run with companion federate.	Compare results log generated by pure-model when executed standalone and when executed with the companion federate.

Table 5-2: Experiment Two Success Requirements

The first criteria is the main focus of experiment two. To be deemed successful, the creation and execution of the AOP-model *must not* necessitate the manual construction of any deployment artefacts. These items require HLA knowledge and this experiment is designed to test their automatic generation (removing this burden from the user).

The second and third criteria both come from experiment one. The second mandates that the pure-model code must not contain any HLA information, as again, this would dictate HLA knowledge on behalf of the user. The final criteria mandates that the results of the standalone and HLA executions must differ, reflecting the fact that remote data plays a role in determining the results for the HLA version.

It is not necessary to retest all operational criteria from the first experiment as the operation of the framework is not the focus this time. The previous experiment validated that those processes worked whereas this experiment seeks to validate that everything still works when an automatically generated object model and set of mappings are used. The successful completion of those tasks is implicit in the meeting of the third criteria. If for some reason any processes suddenly fail, the AOP-model and companion federate would cease to interoperate correctly and the final results for each simulation would reflect this lack of communication. Thus, the meeting of criteria three is depends on all the criteria from the first experiment also being successfully met.

Meeting all the criteria for this experiment means that the same level of functionality that was required in experiment one has been met, except without the use of manual processes that require specialist HLA knowledge. To a large extent, success here in these two experiments realises the goals of this research. The final experiment tests that the developed methods work with an existing federation.

Experiment Three: Existing Simulation Test

Purpose: The purpose of this final experiment is to validate the methods that have been developed and tested through the previous experiments in the context of an existing federation. Thus far, the experimental subjects have been small scenarios, custom designed for use in this thesis. Experiment three involves the creation of a pure OO-model that will interact with an existing implementation of the ATO federation. The major aim of this research is to allow pure-OO models to be developed and automatically rendered as HLA simulation components, capable of being used within live HLA distributed simulation. This experiment is used to further validate that the methods proposed by this research are valid beyond the testing environment used previously.

Prerequisites: The primary prerequisite necessary for this experiment is the development of a pure-OO model that will control an Airport in the ATO federation. There are two final research questions this experiment will help to address. In any situation where a user is attempting to write an OO-model destined to operate as part of a distributed simulation, the tension that exists between a monolithic and distributed environment will be a factor. In a distributed simulation, components must be willing to accept that parts of the calculations are going to occur outside of its boundaries. However, without having specific knowledge of application distribution semantics, this presents a problem.

"How can pure models, that know nothing of application distribution, be created to depend on and work co-operatively with other remote models?"

The previous experiments involved a "legacy" HLA simulation whose primary intent was to log the activities of the federation in order to validate behaviour. In an existing example such as with the ATO federation, co-operative modelling must be undertaken.

Secondly, to be able to operate within the environment of the ATO federation, the pure-OO model must be able to conform to all the relevant federation-level agreements that dictate how the execution of a simulation is managed and how data is exchanged between federates within a simulation.

"Can the definition of federate level agreements be expressed without requiring manual intervention?"
This experiment seeks both to validate that the broad concepts tested previously work in a co-operative modelling scenario and to asses the extent to which comprehensive answers to these additional questions can be developed.

Procedure: The procedure for this experiment is quite straightforward. The pure-OO model must be executed in a federation with live ATO federates. Once again, log files will be collected to ensure the proper operation of the model.

Qualification of Success: The primary signature of success for this experiment is the successful execution of a pure-model within the ATO federation. Previous experiments will have validated the behaviour of the methodology generated in this research. Although log files will be collected to ensure that the proper actions are taking place, success in this context is a rather binary proposition: the model either runs to completion, or it does not (generating errors and crashing). As a primary measure of this, some logging from the pure model will be obtained and visual confirmation of the models affects on the simulation will be captured through the GUI visualisation federate that exists in the ATO implementation being used.

Success Criteria	Validation Method
OO-model runs to completion without error	Manual inspection of simulation run. Supplemented by inspection of log files to validate the lack of any errors
ATO entity information is discovered and used within the OO-model	Inspection of log files to demonstrate that remote data has been found and is active within the pure model
Alterations and actions generated by the OO-model affect simulation state	Capture of visual data demonstrating that aircraft can interact with the pure-OO airport

Table 5-3: Experiment Three Success Requirements

5.3 Summary

This chapter has introduced the shortcomings of AOP when considering its use within the goals of this research. The questions that this work addresses have been raised, and the experimental framework used to validate the generated solutions has been introduced.

This has provided the foundation for the following chapters, which present a discussion of the techniques developed to overcome the problems established here.

Chapter 6

Manual AOP: Separating Model and Platform

Aspect-Oriented Programming provides a methodology that allows developers to isolate crosscutting, system level concerns, thus allowing them to be implemented separately, keeping core business logic free from such considerations. As discussed in previous chapters, the motivations for implementing HLA-behaviour with such an approach is both beneficial and attractive. However, as highlighted in chapter 5, there are a number of serious shortcomings that need to be addressed.

Under the AOP model, it would be common to reuse business logic in a different setting by weaving it into various sets of system-level Aspects. It would also be common for these Aspects to be tailored for their specific environment. With regard to the goals of this research however, this approach presents a significant problem. Applying this approach in the HLA space, one might write a HLA Aspect targeted at a particular simulation. Although this would be useful when attempting to expose new logic to that simulation, it would also be specific to that situation. If a user wanted to expose that same logic to a different HLA simulation, they would require a separate HLA Aspect and weaving rules. Quite clearly, the development of a HLA Aspect would require intimate HLA knowledge, and would thus be unsuitable for this research. It is for this reason that a generic HLA Aspect is necessary to fulfil the goals of this work, and is a prerequisite for experiment one.

This research seeks to define a generically applicable solution to such problems, with the expectation that it will free mainstream developers from the considerable development burden imposed by the HLA specifications. The first step towards achieving this goal is to demonstrate that a sufficiently generic AOP-based environment can be combined with a pure-OO model to create a HLA simulation component (a federate).

In a typical situation where AOP is leveraged, the process of creating the various Aspects and defining how they are woven together is a manual process. Although the overall objective of this research is to automate this process and thus remove the need for any HLA specific knowledge, the first step is to prove that general notion is viable. For this reason, Experiment One (as introduced in section 5.2.3) focuses on the definition of a generic HLA-Aspect that can be woven into pure model code via a manual process. Subsequent experiments (presented in chapters 7 and 8) remove the manual requirement, automating the steps that are deemed acceptable as part of Experiment One.

This chapter defines the proposed structure for the Generic Aspect, describing its inner workings and discussing how it addresses some of the research questions identified in the chapter 5. To conclude the chapter, the results of Experiment One are presented.

6.1 Requirements of a Generic Aspect

For any proposed solution to truly enable the automatic rendering of a pure-OO model as a HLA simulation component, a generic HLA Aspect is necessary. Before investigating the design for such an Aspect, it is important to consider the exact requirements.

6.1.1 Defining "Generic"?

A generic HLA Aspect is one that can be combined with any pure model to create a complete simulation component. That said, in order to function correctly with a particular model, the Aspect must have some knowledge of the interesting information specific to that model, naturally implying some sort of semantic understanding. This leads to a conundrum: what does the term *generic* actually mean in this context?

Primarily, "generic" in this situation refers to a component that is capable of working with an arbitrary model *without* the need for modification. When talking of modification, I am referring to changes to the Aspect at the source code level. The internals of such a component must also remain free of any model-specific considerations, thus allowing it to be portable.

Despite this requirement, for the final simulation component to function properly in a HLA federation, specific information about it is still required by the federate (such as publication and subscription interests, or model-to-SOM mapping details). To remove the need for code-level alterations, this research favours a configurable approach. The provision of model-specific information via configuration data allows the generic HLA component to be used in many situations, yet still have it possess behaviour that is context aware and specific. Viewed holistically, configuration information is still part of the system, and thus can make it model specific. However, when looking at most generic software components in use today, configuration information is the natural way to quarantine and isolate situation specific concerns. RTI implementations are a perfect example of this. They are capable of supporting any simulation model, but this information must be provided to them via configuration data in the form of a FOM.

In AOP, it is the weaving rules that form something akin to configuration information. In most AOP environments, weaving rules are themselves code [5], and as such, require recompilation before use. Although perfectly acceptable, this process does necessitate an extra step to compile the code, a process that could be avoided if separate configuration files were used in the place of hard coded model-specific weaving rules.

6.1.2 Research Questions Addressed

Having established what exactly is meant by the term "Generic Aspect," there are many other services that such a component would also need to provide. Section 5.1 highlighted a number of research questions that identify the shortcomings of a typical AOP approach that must still be addressed.

These questions can be broadly split into two categories:

- The **technical questions** refer to how a proposed solution should behave and react within a distributed simulation

- The **automation questions** refer to how the creation of model-specific items (such as object models and configuration files) can be automated – removing otherwise manual processes

To fulfil its duties, the Generic Aspect solution presented in this chapter must address the first category of questions. With regard to the first experiment, there is no need for answers to the second category of questions. Manual processes are permitted at this early stage.

Of the research questions presented in section 5.1, six fall into category one.

<u>**Object Data**</u>

"How can the creation, removal and alteration of data within an OO model be replicated into an active HLA federation?"

Data in the form of objects and attributes sit at the core of both OO and the HLA. The Generic Aspect must be able to inform the RTI of the creation, modification and deletion of appropriate data within the pure model. Further, when interesting information is created remotely, proxies containing updated values must be created and made available to the pure model.

What exactly defines *"interesting data"* is rather subjective and model specific. In the first experiment, this information would be provided as model specific configuration data.

Data Introduction

"How can the creation, removal and alteration of data within an active HLA federation be replicated within a pure OO model that is not expecting it?"

As mentioned above, the Generic Aspect must make available proxy instances for remotely created data, and in a form that can be consumed by the pure model. How this information can be successfully made available to a pure model depends largely on how the model stores and accesses its own data.

The open ended nature of object oriented programming means there is a myriad of options available to a developer when deciding how to structure, store and access data. In many situations, relevant data is stored in collections (such as lists and maps). Proxy data could be placed directly into these collections in order to make it accessible by the pure model. However, this approach is dependent on the use of single collections within the pure model. In other situations, newly created information might be passed to some model-specific method that then inserts the information in the appropriate places. An example of this is the `enterCar(Car)` method in the Car Race experimental model. Support for introducing data via this type of approach is also necessary.

Data Ownership

"How can the data ownership rules of the HLA be reconciled with the monolithic world-view of pure object-oriented models?"

A monolithic model may attempt to update any piece of data it encounters, even if it is remote data that is not controlled locally. The ownership rules of the HLA only permit this if the model itself either created that information, or has since obtained the specific right of ownership on that information. If pure models are going to co-operate with one another, and with other federates, some method for overcoming this problem is needed.

Interactions and Methods

"How do object-oriented methods translate into HLA interactions?"

In OO, methods form an integral part of behaviour representation. While interactions and methods do not share a complete conceptual overlap, they are alike in many ways. To facilitate the proper and consistent behaviour of a model, some synthesis between these two facilities must be present.

Federate Level Agreements

"Can the definition of federate level agreements be expressed without requiring manual intervention?"

Perhaps one of the single most arduous tasks involved with any HLA simulation is the process of merely getting separate components in synchronisation with one another. Unfortunately, the open nature of federate level agreements makes a complete solution to this problem largely intractable. This chapter outlines the problem of federate level agreements, describing when and where they can or cannot be supported.

Logical Time

"How can logical time be synchronised between a monolithic pure model and a shared distributed simulation?"

Many simulations incorporate a notion of time, even monolithic ones. The rules governing time in the HLA exist to ensure that information is delivered correctly and that the logical ordering of events is maintained. Although the HLA defines a shared notion of time representation, the location and structure of time within a pure model is subjective. Any proposed solution must be able to keep watch on the portion of a model that represents time, only allowing it to be altered in accordance with notifications from the RTI.

Each of these questions must be addressed in the design and methodology employed by any proposed solution that seeks to provide a generically applicable HLA Aspect. Section 6.2 of this chapter begins the account of such a solution, and describes how these questions can be answered.

6.1.3 The Reference Implementation

Before moving on to discuss the precise structure and behaviour of the Generic Aspect solution, it is first necessary to briefly outline the technology chosen to implement these ideas during experimentation (and the effects of this selection). AOP is a methodology, not a technology [56]. Although it is somewhat natural to think of AOP in a low-level technical sense, all AOP tools operate in a manner consistent with a shared methodology. The use of an AOP approach is not specific to a single programming language or tool. This is important when considering the motivations of this work. The sheer number of simulation environments available in the wider business community is enormous. To focus on any one simulation tool would introduce a number of restrictions that are relevant only in that environment.

As such, in an attempt to provide a more broadly applicable solution, this research takes a step back, using Object-Oriented programming as its starting point. Just as there are a great number of tools used for simulation purposes in the wider business community, so too is there a great number of programming languages used for their implementation. The experimental items developed as part of this research make use of one such language, but the concepts and approaches taken are not specific to that language.

For the purposes of testing and experimentation, the Java programming language has been chosen. Java has a long history with AOP, and as such, the tools available are both highly stable and mature. The reference implementation makes use of the open source AspectJ package [5] in order to gain AOP capabilities.

The various tools used were chosen for a number of reasons, primary among which is their open source status. All tools (both HLA and AOP) used in the development of experimentation items make use of open source libraries. Access to the source code has been important in this case, facilitating a deeper understanding of the tools, their structure and function.

The Generic Aspect framework presented in the following sections also makes moderate use of the reflective capabilities of the Java programming language in order to meet its requirements. One potential concern here is that these services are specific to Java, and analogues may not be available in other languages (such as C++). In this work, Java reflection is used primarily as a convenience. Rather then subjecting pure-OO code to greater levels of static analysis, reflection is used to gather the necessary information about a particular set of classes.

In cases where these facilities are not available (such as with non-reflective languages like C++), a custom code parser could be implemented to gain the same information that is extracted via reflection in the reference implementation. Although the development of such a parser would entail a non-trivial amount of work, the same information could still be obtained, and thus, allow the same methods presented here to be ultimately used.

6.2 Simspect: A Generic AOP Environment

The generic HLA Aspect developed as part of this research has been dubbed "Simspect" (Simulation Aspect). To meet the requirements discussed in the previous section, Simspect incorporates both a generically applicable set of weaving rules and a self-contained, intelligent runtime component. Two simple façades isolate the internals of the runtime from the outside, model-specific world. Figure 6-1 is an illustrative overview of the Simspect framework.

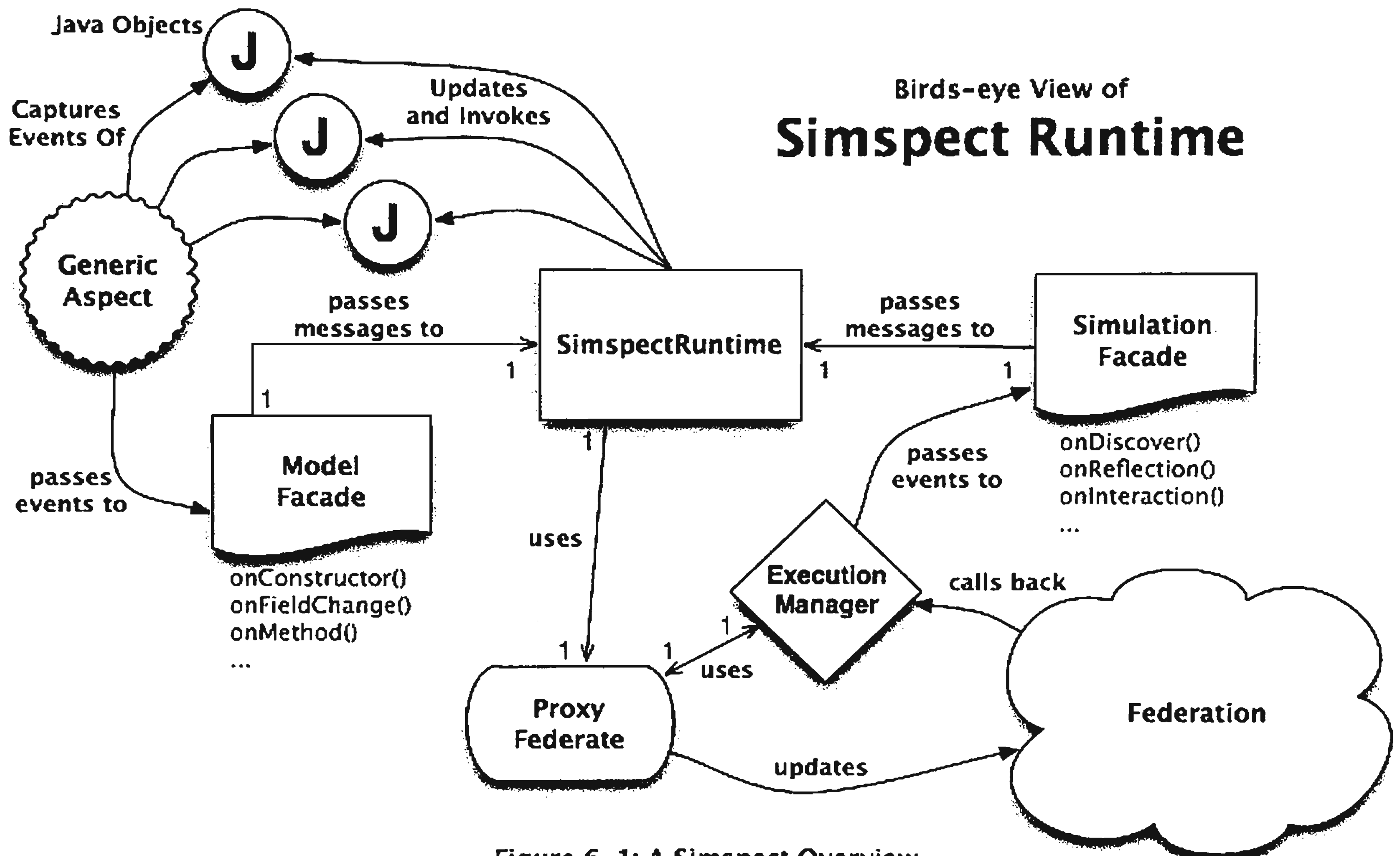


Figure 6-1: A Simspect Overview

As shown in the figure above, the Simspect environment consists of four major components, each of which is discussed in the following subsections:

1. The Generic Aspect
2. The Model Façade
3. The Simspect Runtime (contains proxy and execution manager)
4. The Simulation Façade

6.2.1 The Generic Aspect

In AspectJ terminology, an Aspect refers to both the weaving rules and the advice that should be executed at the captured join points. In a typical AOP scenario, the final step of the development process would be to write these weaving rules and the relevant advice that would make the appropriate calls into the crosscutting Aspects. For example, a weaving rule might capture any call to a constructor of the `Car` class (from the race simulation – see Appendix A) and cause a corresponding HLA instance to be created. These rules, and the advice that is executed when they occur, are designed to be the application specific glue that binds together the various core and crosscutting modules. It is generally thrown away or rewritten when attempting to apply the modules in different circumstances (much like configuration information).

As mentioned earlier, when designing a generic solution, application-specific information must be provided at some point. Weaving rules and advice are one example of how this information could be specified. The approach taken by Simspect however, is a little bit different. Rather than tailoring weaving rules and advice to the specifics of a given OO model, Simspect employs weaving rules that focus on a generic class of events. For example, rather than just capturing the constructor calls for the `Car` class, it captures constructor calls for *every* class. In this way, the decision about the relevance of a model-specific entity is delayed. Rather than defining this information statically at compile time, it is pushed back to be a runtime consideration.

There are advantages and disadvantages to both approaches. When using a catchall solution like totally generic weaving rules, a large amount of uninteresting information that is captured needs to be dealt with at runtime. For example, while the `Car` class may be of interest to wider simulation, the creation and alteration of a logger class would be deemed of only internal interest. As the generic weaving rules cannot themselves identify externally interesting information from data that is for internal model processing, some filtering will need to occur further down execution path. Additionally, as such filtering takes some time; there are potential performance penalties to this approach.

On the other hand, custom weaving rules would remove this problem, ensuring that only relevant information is ever captured. However, this too comes at a cost. To implement such a feature, one must know this information at compile/weaving time. Further, the weaving process will hard-code this information into the final product, necessitating a re-coding and recompilation step for even a minor change.

The primary benefit of the "catch-all" approach is that a single set of weaving rules (and associated advice) can be employed with any pure-OO model, regardless of its structure. By capturing all constructors, there is no need to have prior knowledge of what is and is not a relevant piece of information. The authoring of Aspect weaving rules and advice necessitates knowledge of both AOP and the syntax of the particular framework in use. Although the "separation of concerns" concept is quite simple to grasp, the low level details of weaving together core and crosscutting concerns in a manner suitable to a particular AOP framework is perhaps unnecessary for most users. Once again, the use of generic weaving rules means that they only need to be defined and written once, rather than over and over again for every different model.

For all of these reasons, Simspect uses generic weaving rules and advice, relying on configuration data to provide the model-specific information necessary to filter out unnecessary information at runtime.

The Generic Weaving Rules and Advice

Having settled on the use of generic weaving rules and advice, attention must be turned to exactly what type of events these rules are going to capture. Rather than focusing on the model-related entities themselves, the Simspect Aspect seeks to capture typical classes of events that occur during the execution of any OO application.

The Simspect Aspect captures the following types of events:

1. Main Method Invocation
2. Constructor calls
3. Field alterations (both instance and class fields)
4. Method Calls
5. Object Removals

As one might expect, each event class represents a rather broad concept. What may be surprising is the size of the list, with only five types of events holding any interest. Despite this, the list represents the full gamut of information needed by Simspect to perform its duties. Despite the considerable complexity of the goals of this research, the specification of the weaving rules is remarkably simple, consuming fewer than 35 lines of code in the reference implementation[10].

The generic advice that is executed for each of the weaving rules above is equally straightforward. Its primary purpose is to pass information about the event on to the Model Façade (discussed in the next section) where it can be filtered. To illustrate this point, consider how object construction is handled.

The specification of the weaving rule is as follows:

```
/** pointcut to get all constructors */
protected pointcut constructors( Object newObject ) :
        initialization( public *.new(..) ) &&
        ignoreList() &&
        target( newObject );
```

Listing 6-1: Constructor Point Cuts

This statement basically says: "capture every call to a *public* constructor on *any* (*) class, as long as that class is not on the ignore list". The final line is necessary to capture the context at the join point [23] (in this case, the object being created).

The advice associated with this weaving rule is perhaps even simpler than the weaving rule itself:

```
before( Object newObject ) : constructors( newObject )
{
        // notify the runtime //
        this.facade.onConstructor( newObject );
        logger.debug( "{NEW} " + newObject.getClass() );
}
```

Listing 6-2: Constructor Advice

This advice consists of a single line of effective code (and some logging), passing the information about the constructor to the model façade. How Simspect reacts to this information (if it reacts at all) is of no concern to the Aspect. Its purpose is to capture the

[10] A listing of the generic weaving rules and advice is provided in Appendix D

information and pass it onwards to the façade which can then respond appropriately. Not all the events captured by the Aspect involve a single action. The table below outlines what happens for each event.

Event	Action
Main Method	1. Instantiate Model Façade 2. Inform façade that model is starting 3. Proceed with main method (model executes) 4. Inform façade that model is finished
Constructor	1. Inform the façade
Field Alteration	1. Inform the façade 2. If response from façade is `true`, proceed with field alteration, if it is `false`, skip the alteration
Method Call	1. Inform the façade 2. If response is from façade is `true`, proceed with method, if it is `false`, skip method execution
Object Removal	1. Inform the façade

Table 6-1: Model Event Actions

As shown in this table, with the exception of object removal, the other events include additional steps. These additional steps may also be a point of some confusion. The main method event is known as an "around" advice [57] and it has the effect of wrapping the execution of the main method and allowing advice to be executed both before and after the main method call. If the advice so decides, it can even skip the call to the wrapped method altogether.

The method-call and field alteration events also use "around" advice. In these cases, the façade is notified of the event, and a value indicating whether or not the event should proceed is returned. This is useful in situations where requests must be ignored because they relate to remote information. For example, when a method is being called on an object that was not created locally. In a typical HLA simulation, changes to remote data are subject to the processing approaches used in the models that have ownership over that data. By introducing new models that contain different processing rules, sets of objects can be updated via differing approaches. For example, a `Car` created in a remote federate could implement a different advancement algorithm to the `Cars` that were created in other federates (or indeed, those created locally). In order to allow this polymorphism, methods called on remote objects are quietly discarded by the runtime when they are called. The return value provided by the façade indicates to the advice whether or not the underlying method call should proceed. If the data is local, there is no concern and the method can proceed as normal. This process however places the specifics of that decision beyond the façade, allowing the advice to have no knowledge of how the decision was reached and to remain quite simplistic.

Weaving Exclusions

It is worth highlighting that within the reference implementation there are certain exclusions placed on each of the weaving rules that capture these events. One might expect that in any non-trivial model there will be thousands of constructor calls, and many times more field modifications and method calls. Many of these will be to default class libraries and will be of no interest to the model at all. Exclusions act much like a veto, ensuring that advice is not woven into a model if the class in question happens to reside in a certain Java package. For example, no advice will even be executed for constructors, method calls or field alterations on classes that make up the core Java class library. Further, any code within Simspect itself is also excluded from these rules (as is code from the libraries used by the framework). The interesting parts of a model are provided in the model code itself, and these exclusions are a simple way of ensuring that the focus remains purely on that logic.

Simple Yet Effective

The purpose of the Generic Aspect is simple: capture the events and notify the façade so that it may trigger the necessary processing required and take action. Isolating the Aspect and the processing logic in this way helps to keep the Generic Aspect both small and straightforward.

6.2.2 The Model and Simulation Façades

The façade is a well-known design pattern [38] implemented in an attempt to lower cohesion between two components by hiding potentially complex actions behind an opaque interface. Figure 6-1 identifies two façades within the Simspect environment: One for model events and the other for simulation events. Much like the Generic Aspect, the actions taken by the façades are exceedingly simple. They exist primarily as something akin to a traffic policeman. When an event occurs, they pass the information on to the Simspect runtime, which then does all the necessary processing and informs the façades of the results.

As is discussed in the next section, the Simspect runtime does little but provide a superstructure into which event information can be dropped. Internally, the façades follow the *Command* design pattern [38]. This pattern describes a situation where event information is wrapped up inside an instance of a special class. This instance conveniently encapsulates all the relevant information about the event in a single entity. In Simspect, these instances are generically referred to as *Messages*. The façades are responsible for creating the message objects (from the information they receive) and passing them into the Simspect Runtime. This is not a particularly involved process, generally involving only one or two steps.

To once again illustrate this simplicity, consider the actions taken by the Model Façade when it is informed of an object construction:

```
public void onConstructor( Object object )
{
        // create and process the message //
        fireMessage( new MDL_OnConstructor(object) );
}
```

Listing 6-3: Model Constructor Notification

The code above is taken from the Model Façade in the reference implementation. In this listing, the `MDL_OnConstructor` class is the specific message type. It is given all the relevant information, and then the `fireMessage()` method informs the Simspect Runtime[11].

The Simulation Façade provides simulation relevant functions for handling any information received from the RTI. It behaves in exactly the same manner as the Model Façade, packaging all the relevant information up into message types and passing them into the Simspect Runtime for processing. As discussed in section 6.2.3, each runtime contains a proxy federate that sits inside an active federation and represents a pure model. While the Model Façade is invoked via AOP advice when relevant model events occur, the Simulation Façade is notified via the Federate Ambassador of the proxy federate. When the RTI provides information to the proxy via an ambassador call-back, that information is then passed directly to the Simulation Façade where it can be packaged and sent to the runtime.

[11] The `fireMessage()` method performs some general housekeeping. It packages the message into a special container type (which includes room for a response). It also serves to extract the response once the runtime is finished, and performs some basic error handling.

The façade classes are simple components. Beyond packaging request information into an appropriate form and passing it to the runtime, they play no further role in the processing of model or simulation events. The responsibility of acting on the events and performing the necessary actions is delegated to the Simspect Runtime itself.

6.2.3 The Simspect Runtime

The runtime represents the nerve centre of the Simspect framework. To this point, discussion has focused on how events are captured (the Generic Aspect and proxy federate) and delivered (through façades) to the Simspect Runtime. This subsection briefly introduces the components and internal structure of the runtime framework itself.

In isolation, the Simspect Runtime is capable of very little. Rather, it provides a superstructure into which specialised processing units can be inserted. These units, known as *Message Handlers*, are responsible for performing the appropriate actions when the runtime is notified of an event via a message from the façades. The message handlers can be seen as the drivers of the Simspect Runtime, using other parts of the framework to perform any necessary actions. Figure 6-2 shows the internal structure of the Simspect Runtime. The role of each component is discussed below.

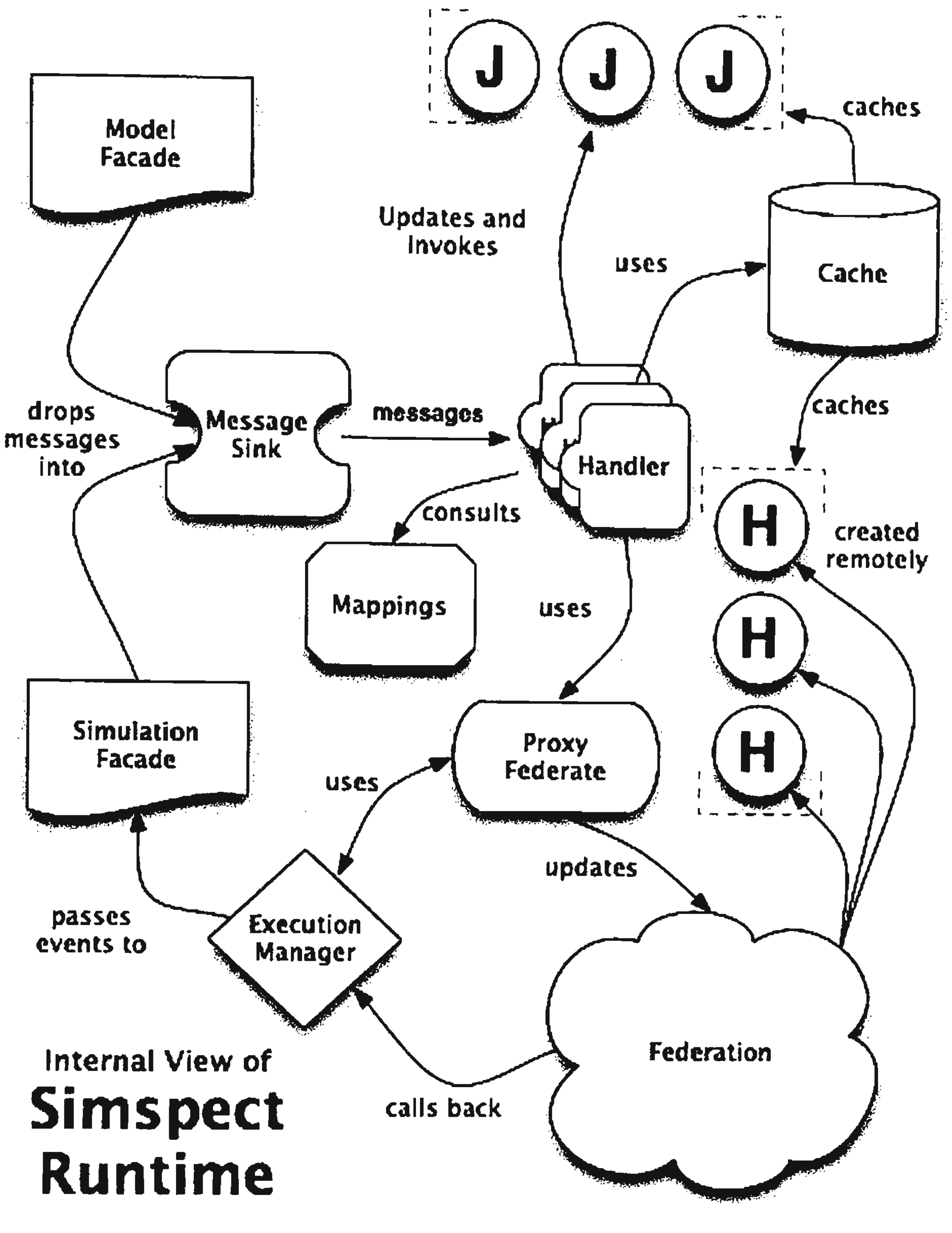


Figure 6-2: Simspect Internals

The Message Sink

As figure 6-2 shows, each of the façades passes message instances directly into the *Message Sink*. The role of the sink is to aggregate together a group of handlers; recording which messages a given handler is interested in processing. When the sink is given a message, it consults its internal registry and finds the handler associated with the type of message that has been received. The message is then passed to the handler so that it may perform any necessary actions. Each handler has a dedicated task to perform and may make use of some other components (such as the Cache or Mappings information) to this end.

The entire process is very similar to both the *Observer* design pattern [38] and the publish/subscribe facilities described in the HLA specification itself. There is no outstanding requirement that dictates the Simspect Runtime must take on a processing form such as that described, but the command/observer pattern mix does provide considerable advantages.

Primary among these advantages is the ease of adaptation and extension this approach provides. Should the runtime at some point need to process new types of events, additional handlers can be readily inserted directly into the framework without necessitating extensive refactoring. Further, should the requirements for a handler change in any way, the effect of this is limited to that handler. This is of particular interest when considering the reuse of this design in a programming language or environment that differs from that of the reference implementation.

For example, the reference implementation handlers make considerable use of reflection, a dynamic introspection facility provided by the Java platform [35]. In other languages (such as C++) this mechanism is not immediately available, necessitating an alternate approach to be used (as discussed in section 6.1.3). The important point is that regardless of how that information is accessed, the general runtime structure does not need to be altered in any significant way. This information can be localised to the specific handlers that use this information, while the Aspect, façade, proxy and so forth all remain as defined here.

As mentioned above, the handlers represent the interesting portion of the reference implementation and define solutions to the problems referenced in the research questions. The methods used by these handlers to address the problems raised earlier are discussed in detail in section 6.3.

The Cache and Mappings

The *Cache* and *Mappings* components depicted in figure 6-2 are utilities used by the various handlers when completing their tasks. They store both useful information needed by the runtime to determine whether or not action should be taken, and, links to the relevant data on which operations should occur.

Mappings

As discussed in section 6.2.1, the weaving rules and advice are entirely generic. They capture events that are *potentially* interesting, rather than restricting this to only those that are *known* to be interesting for a specific OO-model. These captured events need to be filtered at some point so that only those of some consequence to the remote federation are acted on (while the rest are ignored and deemed "internal model processing"). The *Mappings* component was defined as a container for this critical information.

Filled from configuration data, each entry in this component describes how a particular object-oriented class maps to its HLA-based counterpart. From this list, the relevance of a particular class can be quickly determined: if it has no entry, it is of no interest. This enables the required filtering to occur when OO-model events are received. It is important to note that the same functionality is not required for received HLA data. Through the declaration of subscription interests, the RTI will only deliver to the proxy federate data it has previously signalled an interest in. In this case, the RTI performs the filtering on the runtime's behalf.

Mappings data is not used purely for filtering purposes; rather, it has many other roles. As mapping data defines how OO-model constructs relate to their HLA counterparts, it is also useful in determining the publication and subscription interests of the proxy federate. The exact way this data is used for this purpose is discussed in section 6.3.

Mappings data also contains transformation details. As the HLA represents data as an opaque series of bytes, some knowledge about the intended structure of those bytes is necessary. Mappings data provides this. The transformation of data between the model of a specific simulation component (SOM) and the shared model of a federation (FOM) has been the subject of considerable prior research (see section 4.2.1). Rather than define a solution to the complex issues surrounding this topic, this research defers to that work, whose approaches could be implemented inside the Simspect Runtime as required. That said, the reference implementation requires some basic transformative capabilities, and the mappings data provide this information. Unlike the previous work, these capabilities deal only with simple, primitive types (such as strings, integers and floating-point numbers). In situations where a more robust solution is required, the work highlighted in section 4.2.1 should be drawn on.

Cache

Unlike mappings data, the Cache is populated at runtime. It links together instance information that exists in the OO-model with the parallels that exist in the HLA. In figure

6-2, as with figure 6-1, the circles containing a "J" (for Java) represent local information, while HLA data items contain a "H". For data that was created locally (by the proxy federate) in response to an object constructor call, the cache maintains a link to the OO object instance in addition to other information the proxy federate will need (such as the object handle). For data that was created remotely and discovered through the RTI, this same information is maintained, along with the last known values that were received as part of an attribute reflection for the instance. The way this data is created and used is detailed in section 6.3.

The Proxy Federate

The Proxy Federate is the gateway to the HLA for a pure-OO model. Each runtime instances contains a single proxy federate through which engagement with the HLA is achieved. When the handlers deem it necessary to forward information to the HLA, it is routed through the proxy federate. When information is received from the RTI via the Federate Ambassador for the proxy federate, it is passed directly to the Simulation Façade, where event information is created and dropped into the message sink.

The Execution Manager

The Execution Manager is a sub-component of the proxy federate. The problems of federate-level agreements were highlighted at the beginning of this chapter, and the Execution Manager is a direct response to some of those concerns. The main difficulty of federate-level agreements is that there is no standard approach for defining or describing them. Any federate can define rules for how they behave within a federation, and any such combination of actions is perfectly valid. The execution manager is the facility through which execution management federate level agreements can be specified.

Furthering this problem is the lack of a parallel for execution management within OO. Once again, the differences between a monolithic pure-OO model and a co-operative, distributed simulation are a chief cause of this disconnect. These agreements are an artefact of the HLA and remain entirely in that realm. In practice, the only way to overcome such issues is through the standardisation of a particular execution model. Regrettably, no such standard exists and this problem persists. Despite this, some recourse for addressing these issues is necessary.

The Execution Manager component is the facility through which different execution methods can be supported. When certain simulation events occur, it is notified and may take any appropriate execution management actions necessary. These events include:

- **Synchronisation Points**: announcement, registration, federation synchronisation
- **Time Advancement**: time advance granted
- **Data Received**: interaction received, object discovery/reflection/removal
- **Save and Restore**: save/restore initiated, started, achieved

These types of events cover the full spectrum of call-backs from the HLA, any of which could contain information pertinent to execution management procedure. The Execution Manager for the runtime can be easily replaced, allowing different execution models to be supported without affecting the entire framework. A default manager should be provided with any implementation of the Simspect runtime, thus only requiring a new one to be authored when OO models must interact with existing, non-compliant HLA simulations.

Naturally, the development of an Execution Manager implementation would require HLA knowledge and skills. While this requirement is generally not tolerable within this research, in this particular situation it is unavoidable. That said, this would be limited to the execution of OO-models in specific scenarios, and would not affect model development.

Much like mapping data, the specification of the relevant Execution Management implementation used by the runtime is defined through configuration data (discussed below).

Unfortunately, there is no genuine solution to the problem of broader federate level agreements (such as data structures or attribute update intervals). While the HLA 1516 standard does help with some classes of these problems, the open nature of the agreements means that there is no generic solution to the problem outside of mandating that the agreements be based on well-accepted conventions.

6.2.4 Customising Simspect Through Configuration

Earlier in this chapter the generic nature of the Aspect weaving rules and advice was discussed. In that section, it was conveyed that the approach taken by the solutions presented here was to push the customisation for a particular situation into configuration data rather than the rules that capture information themselves. This has the benefits of being adaptable without necessitating re-coding of the framework, and also allows additional information (such as HLA related data) to form part of the configuration, whereas AOP weaving rules and advice only relate to the pure model.

In the previous sub-sections the notion of the *Mappings* and *Execution Manager* were introduced. Each of these components depends on information provided as part of the configuration data. This section briefly introduces how this data is provided to the Simspect runtime, as it exists in the reference implementation used for experimentation.

Figure 6-3 highlights the process involved in the creation of the runtime and the reading of configuration data.

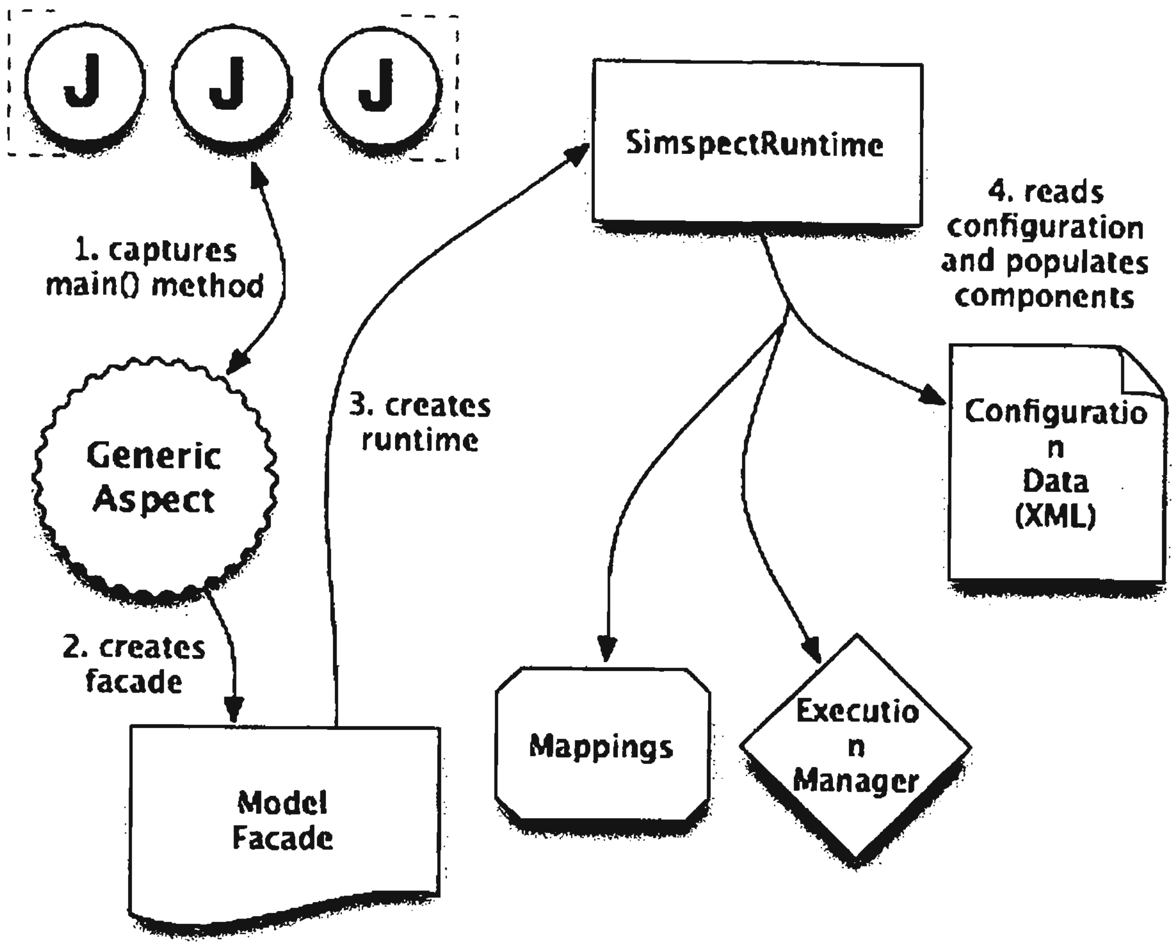


Figure 6-3: Simspect Configuration

When the Generic Aspect realises that the OO-model is about to begin, it triggers the process that creates the entire Simspect runtime. At this point configuration data is obtained from an XML document and used to populate the Mappings and create the appropriate Execution Manager implementation. In this way, the data interests of the runtime, or the concrete implementation of the Execution Manager can be manipulated without the need for authoring any code within Simspect. The runtime remains entirely generic, with the customisation data that ties it to a specific OO or HLA model being gathered from an external source. Within the context of the first experiment (discussed in section 6.4) this configuration data is generated manually. Further experiments focus on automating the generation of this information following the inspection of a particular OO-model.

6.3 Handler Methodologies

The previous section introduced the structural design of the Simspect runtime. This section talks about the various approaches and methodologies used by the handlers inside that framework that address the research questions highlighted at the beginning of this chapter. This will explain how those questions are addressed, leading into a discussion of experiment one, which validates the approaches defined here. The following subsections are broken down by research question, with each one discussing the various approaches taken for addressing it.

Throughout this section, various parts of the reference implementation and underlying platform are discussed directly. This is done purely to provide some context to the discussion. It is important to keep in mind that the methodology employed in the reference implementation is the important part of the discussion, and that these ideas can be implemented in alternate settings, not just with the technology leveraged in the reference implementation created for experimentation.

6.3.1 Composite Objects and Complex Data Types

As first introduced in section 3.2.2, the HLA does not represent data in the same manner as object-orientation. Although similar notations (such as inheritance) exist in both domains, on the whole, there are significant differences that must be considered. Primary among these is the representation of complex data types.

Unlike object orientation, HLA object models do not support the expression of explicit relationships that may exist between the instances of various classes [62]. For example, there is no standard mechanism within the HLA for defining that a relationship exists between an instance of a `Wheel` class, and an instance of a `Car` class. Although they are intimately linked conceptually, associating one instance with another requires explicit action on behalf of a federate developer.

The IEEE 1516 specification provided considerably more powerful support for expressing the structure of complex data types [49]. Complex structures could be defined, and attributes would be declared to contain data conforming to those types (as is typical in OO). Despite enabling composition, this advancement came at a cost.

A primary feature of the HLA is its support for defining interests in data at the attribute level. For example, a viewer federate may subscribe to only the `distanceTravelled` attribute of the `Car` class. Enabling publication and subscription to occur at higher levels of fidelity removes any need for the transmission of redundant data, with only that information that has been explicitly requested by a federate being delivered to it. However, if `Car` were to be defined as a complex data type, this ability would not exist. A particular attribute could be a `Car`, but as subscription works at the attribute level, there would be no way to subscribe to individual pieces of the car. IEEE 1516 data types exist only to provide some information on how to decode opaque values associated with attributes, and as such, do not allow for the individual pieces of the type to be considered.

The situation becomes something akin to "all-or-nothing," resulting in the potential communication of redundant information, and the inability of federates to co-operatively model the constituent parts of that type. It is for this reason, that the use of such types has been avoided in this research. Rather than reducing the fidelity of the model itself, another approach for representing composition relationships is needed.

Fortunately, this problem can be solved via simple substitution. When an attribute is to aggregate another object instance, the value of that attribute should be a reference to the instance. This is the same approach used in virtually every modern programming language, where pointers or memory references are used to link together otherwise separate structures. While an OO approach would support this mechanism at the language level, it is unfortunate that within the HLA the federate developer must handle it manually.

Throughout the Simspect framework, wherever an aggregation or other such reference (such as in a parameter to an interaction) much be used, a substitution of actual instance data for it the objects federation-wide unique handle is transparently made. Thus, it would appear to the pure model as if a reference to the instance were passed, rather than an object handle.

<u>Collection Representation</u>

Collections types are used extensively in typical object-oriented model development. Grouping together multiple instances, they provide a nice way to represent many objects in a single reference. Although the IEEE 1516 specification introduced support for defining arrays of data types, those advancements are being overlooked by this research for the reasons mentioned above.

However, once one has accepted that object references are to be represented as their HLA object handles, the transfer of collection information can also be easily solved. A collection is a sequence of objects, thus, when communicating via the HLA, the Simspect runtime represents collections as a series of object handles. The byte value transferred for an attribute that represents a collection is equal to the byte value of each handle in the collection, stored end-on-end. As with single object references, it would be expected that any Simspect runtime implementation would transparently transform each handle into a reference to its associated local object, and each series of handles into a collection of those objects.

Subsequent sections make little, it any reference to the manner in which object references are handled. The methods described here are used throughout the processes discussed over the remainder of this chapter. Notably, the definition of this approach ends up constituting a federate-level agreement, and for pure models to co-operate properly with HLA-only simulations, those simulations would need to conform to the expectations outlined here.

6.3.2 Object Data

One of the most obvious problems facing this research is how information is extracted from an OO-model and pushed into an active HLA simulation. The previous sections of this chapter have covered how information is extracted from a pure model (via the Generic Aspect) and sent to the runtime. This section discusses what happens after that point.

There are three major events of interest that overlap between a pure model and a distributed simulation: object creation, field updates and object removal. Fortunately, parallels for each of these events exist in both domains.

<u>Object Creation</u>

Object creation is triggered when the runtime is informed that a constructor has been called. The primary decision that must be made is whether or not this creation should be relayed to the active federation. Figure 6-4 is a flow chart describing the process involved in making this decision.

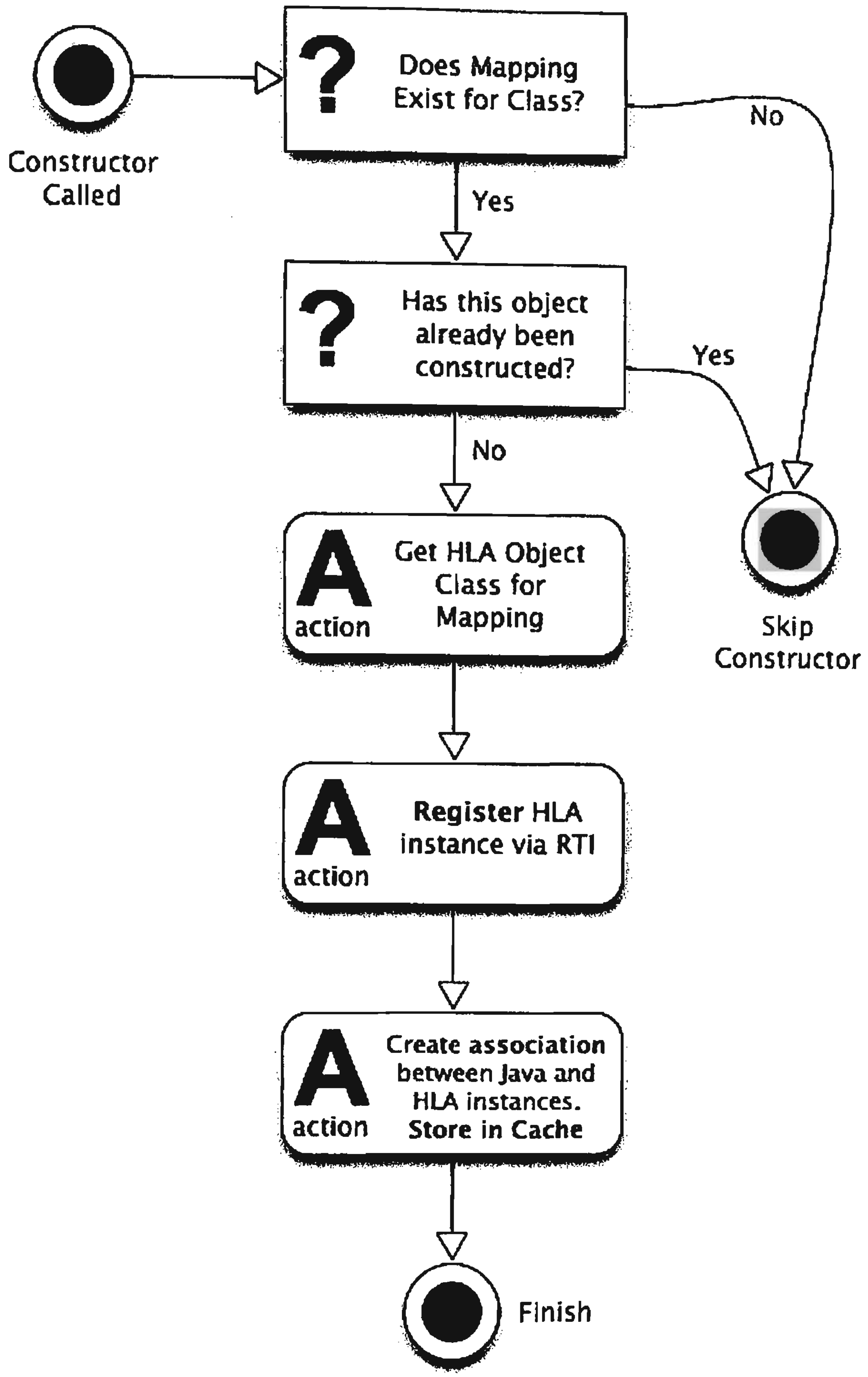

Constructor Called
Does Mapping Exist for Class?
No
Yes
Has this object already been constructed?
Yes
No
Skip Constructor
action
Get HLA Object Class for Mapping
action
Register HLA instance via RTI
action
Create association between Java and HLA instances. Store in Cache
Finish

Figure 6-4: Constructor Called Flowchart

The first decision that must be made is whether or not the class of which an instance is about to be constructed holds any interest within the connected federation. Put another way, does the class have a parallel in the FOM? Within the Simspect framework, the process of determining this is quite straightforward. If the class is of some interest, it will have been included in the configuration data that was read by the runtime when it started up. Thus, if there is *not* an entry in the Mappings entity (see figure 6-2) for the class in question, then it is merely internal data. It is this step that performs the filtering necessitated by the use of an entirely Generic Aspect (discussed in section 6.2.1). If there is no mapping for the class, the constructor is ignored (filtered out) and processing of the model continues. If there is a mapping for the class, then the information about the event must be relayed to the federation.

Following this, another decision must be made, although this one seems slightly confusing. It must be determined whether or not the instance has previously been registered. Initial thinking might believe one to think that each instance can only be constructed once, and thus, only a single constructor event will ever be invoked once for each instance. Generally this is true, but this is not always the case when inheritance and super classes are involved. I will return to this question shortly, but for now, let us just assume that the instance has not previously been registered and that processing can proceed.

If this is so, the relevant HLA Object Class handle is extracted from the Mapping entry[12] and is used to instruct the Proxy Federate to register an instance. Once this has been completed, an **association** is made between the HLA instance information and the Java instance. At the same time, an additional **field association** (which is a sub-component of the general association) is made between all the fields identified in the Mapping (those of interest to the federation) and the fields of the model instance. These associations contain additional information (such as the HLA handles of the various objects, classes and attributes) and are used later as a means to identify which HLA instance to update when a field value is modified for the associated Java instance. This association is stored inside the Cache for later access. Once this process is completed, execution returns to the pure model.

Initial Instance Variable Values

The ordering in which events occur here is important. The AOP advice that captures constructor events such as this is defined as "before" advice. That is, the advice (and thus

[12] During start-up, the Mappings are resolved against the FOM to obtain the various handles as they are defined for that particular federation execution.

the handler) will execute *before* the body of the constructor call. It is only once the advice has completed that the body of the constructor is executed. This guarantees that the HLA instance will be registered via the Proxy before the model instance has finished being created.

In most constructors, some initialisation of instance fields takes place. It would appear reasonable that these initial values be reflected into the federation as they are created. The same pointcuts that capture general field modifications will also capture them in this setting. As such, the HLA instance needs to exist first, for if it did not, the runtime may attempt to update the attribute values of a HLA instance that is yet to exist. By creating the HLA instance first, this situation is avoided, and the process of handling the initial setting of instance variables can be managed in the same way as any other field modification.

Super Constructors and Constructor Chaining

In any OO language, inheritance is a pivotal feature. Encapsulation is also a central tenet of object orientation, stating that only a specific class should modify its own data. Following this notion, it does not make sense for the constructor of a given class to modify the variables it inherits from a super class. Thus, it is common for the constructor of a class to first call the constructor of its parent class. The problem with this is that AOP recognises this as another constructor event, which is captured and sent to the façade.

If the parent class also happens to be of interest within the HLA federation, this approach has the potential to register additional HLA instances for the same model object. This is clearly an unacceptable situation. Some mechanism is needed to identify this situation and only proceed with registration the first time around.

AOP provides a mechanism for identifying whether or not a call is in the "control flow" of another call. This approach may have some potential to address this problem by redefining the pointcut such that if the constructor is in the control flow of another constructor, it does not get captured. However, only further inspection, this approach is also not suitable. Consider the following code snippet:

```
public Restaurant( String name )
{
        super( name );
        this.vipTable = new Table();
}
```

Listing 6-4: Restaurant Constructor and Control Flow

If control flow were to be used to ignore any constructors within the `Restaurant` constructor, it would also omit the call to the `Table` constructor. If the `Table` class were of interest to the active federation, this would mean that valuable information is not made known. Thus, using the AOP control flow mechanism is not suitable.

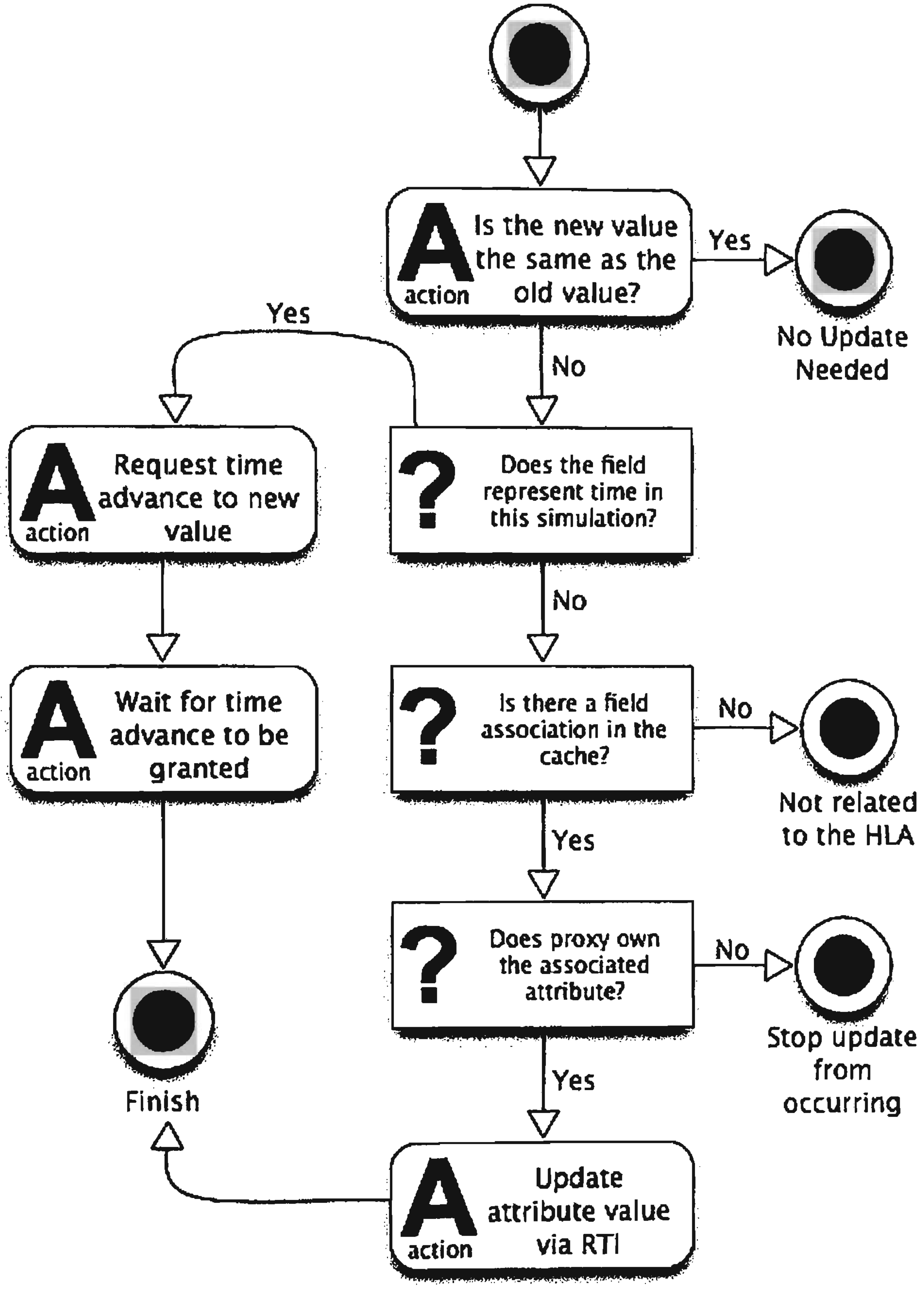
The key to solving this problem is to recognise that only a single instance is created in the pure model, regardless of how many constructors are called for it. As mentioned above, once a HLA instance has been created, an association is made between it and the model instance. This association is then stored in the Cache. When the constructor handler is invoked, it is passed a reference for the instance that is being created. Thus, the handler can check the Cache to see if an association for this reference already exists. If it does, the handler knows the event is part of an inheritance chain and that it should be ignored.

This approach allows constructors for other classes to still execute as part of a constructor (as in the snippet above) but ensures multiple HLA instances will not be registered for a single model instance.

Field Modification

Field modification events are triggered whenever some piece of code updates any instance variable. The decision tree controlling how field modifications are handled is considerably more complex than is the case for constructors, with many alternate processing routes needing consideration.

Figure 6-5 outlines the decision process used when a field modification event is handled:



Is the new value the same as the old value?
action
Yes
No Update Needed
No
Yes
Request time advance to new value
action
Does the field represent time in this simulation?
No
Wait for time advance to be granted
action
Is there a field association in the cache?
No
Not related to the HLA
Yes
Does proxy own the associated attribute?
No
Stop update from occurring
Yes
Finish
Update attribute value via RTI
action

Figure 6-5: Java Field Modified Flowchart

The first decision made when the handler receives a field modification event is used primarily for efficiency. On occasion, a field may be "updated" with a new value that is exactly the same as the old value. In these scenarios, the handler implements some basic filtering, and no reflection is sent out. If the new and current values are not identical, the real processing begins, starting with determining whether or not the field represents "time" inside the pure model. The handling of logical time is discussed fully in section 6.3.5. For now, assume that the field being set is not the same field used to represent time in the simulation.

Just as with constructor events, a determination must be made about the relevance of the field that is being updated. Although a particular class of object may be of interest, it is probable that only a small number of attributes within that class are of relevance. The other variables may pertain to internal processing data, or information that simply has no importance in the broader context of the distributed simulation. The semantics of determining whether or not a field is of interest is non-trivial, taking into account a number of characteristics. Fortunately, in the case of the Simspect runtime, the Cache provides an excellent facility for filtering.

In the discussion of constructor handling, it was mentioned that when an instance is registered with the HLA, associations are made between each field of interest (as defined by the mappings) within the Java object, and the cached HLA value. If the Mappings define that an attribute is not of interest, no such association is made.

The Cache is asked to return the association that links the Java attribute for the particular instance being updated with its HLA equivalent. There are three possible situations arising from this request:

1. The Cache does not recognise the object, meaning no HLA instance has been registered for it (perhaps because it is not of an interesting type)
2. The Cache finds an association for the object, but it does not contain an association for the attribute in question. While the object is of a type that contains *some* interesting attributes, this particular attribute is not relevant
3. The Cache finds an association for the instance, and inside it, an association for the attribute

If no association exists (as in 1 and 2), the field is not of any interest to the HLA and no action is taken. At this point, the modification of the field value inside the model is still allowed to complete, but no reflection is sent.

When a field association has been located, the new value can be reflected into the federation. However, the process that covers how this happens depends on whether or not ownership of the attribute rests with the Proxy Federate.

As previously noted, a pure model makes no distinction between owned and unowned data. Being monolithic in nature, it will attempt to update any field value at any time. If the Proxy Federate has ownership of the relevant attribute, the new value is reflected out to the federation, and the field modification within the pure-model is allowed to proceed. However, if the data is owned by a remote federate, no update is sent (as it would result in an error from the RTI) and the field modification in the local model is disallowed by causing the around advice to skip the actual field modification. To understand the reasoning for this, one must investigate the potential approaches for updating unowned attributes.

Updating Unowned Attribute Data

Altering the value of an unowned attribute is a problem created by the HLA ownership model. It is a problem that can be solved in a couple of alternate ways. Unfortunately, common approaches fall into the realm of federate-level agreements, requiring understanding and support from other federates to succeed. The two general methodologies are: "ownership acquisition" and "update requesting".

Under acquisition, ownership over the particular attribute is passed from one federate to another. In this case, the Proxy would explicitly request ownership, and the current owner would need to be programmed to divest this permission. Once ownership has been transferred, the receiving federate can perform the update. The single advantage of this approach is that it allows the federate to perform the update itself. However, there are a number of problems.

Firstly, it requires the co-operation of other federates, and the ownership facilities of the HLA are generally ranked among those least used and supported. Secondly, it raises questions about what is to be done with the attribute once the update has completed. Should ownership be returned to the original federate? Should the new federate maintain ownership? This is all federation dependent, and extremely difficult to support in any generic fashion.

Another common solution to this problem is to have the federate directly request an update of the attribute, generally through some predefined interaction. When a federate wishes to update an unowned attribute, it issues the interaction with all the relevant information and relies on the owner to oblige. This approach shares the primary drawback of ownership acquisition in that it depends on other federates recognising the situation and behaving according to some federate-level agreement. On the surface, neither of these approaches appears desirable, generally for the same reasons. However, when considering how object-oriented models behave, somewhat surprisingly, there are compelling parallels to be drawn.

Consider the object-oriented approach for attribute value updating. Generally speaking, it is considered poor form for the instance variables of a particular class to be accessed and updated externally. If another class wishes to institute a change, the accepted convention is that this is done via a special method, known as a *mutator* [28] (or a *setter*). This is a method of a class that accepts a new value and applies it to the contained instance variable. This approach allows any manner of error checking or other necessary logic to be executed before the variable is updated. Further, in allows the representation of the value to be encapsulated. For example, a single external property may in fact be implemented as two separate variables. The mutator convention is usually supported by marking all instance values as private, or not accessible except from within the functions of the associated class.

When considering such an approach, a type of pseudo-ownership begins to emerge. Although at a high level, the data is still considered part of the same model (and thus there is no inter-model or distributed ownership), there is an element of the update request methodology to this process. A request to update an instance variable is made from a function external to the class that contains it. The mutator method of that class can then decide whether or not to permit this change, and how to alter the internal representation.

In a HLA context, the external class could represent the pure model that wants to update unowned data. The associated class would be the federate that owns that data. Whether or not the request is honoured is entirely the decision of the owning federate. Further, how the update affects the object's state is also controlled by the owner federate. This is the essence of the co-operative modelling approach that is a pivotal advantage of the HLA.

It is through this line of inquiry that the request/update method appears to represent the closest semantic fit between the two distinct development models. While incompatibilities will exist when a federate is not programmed to reciprocate to such requests in the expected manner, these edge cases are unavoidable, and are equivalent to an OO-model including a misbehaving class.

Having established that the request/update approach (although flawed) best suits this particular situation, it may seem surprising that the interaction-based solution presented earlier is *not adopted* in the decision tree of figure 6-5. Rather, all requests to update unowned attributes are ignored by the runtime handler, and it forbids such updates from proceeding in the pure model. As the field modification is an *around* advice, the handler causes false to be returned from the façade, which the advice interprets as instruction to not proceed with the modification.

While this solution could incorporate an explicit update request interaction as suggested earlier, there is no need. All OO-models should update information through mutators[13], and as such, supporting those requests falls into the same category as supporting methods calls in general. Given this, requests to update unowned attributes are silently discarded, deferring handling to the mechanism that supports methods. Method handling is discussed in section 6.3.3.

Object Removal

Handling object removal is perhaps the simplest of all processes. The process consists of only a single decision, which determines whether or not there is information worth removing. Figure 6-6 shows the flowchart describing this:

13 Although models can violate this requirement quite easily in any object-oriented language, it is a requirement of this work that they do not. There is any number of ways an OO program could be constructed that would render it incompatible with any solution presented. Thus, consideration here is restricted to models that adhere to standard OO conventions.

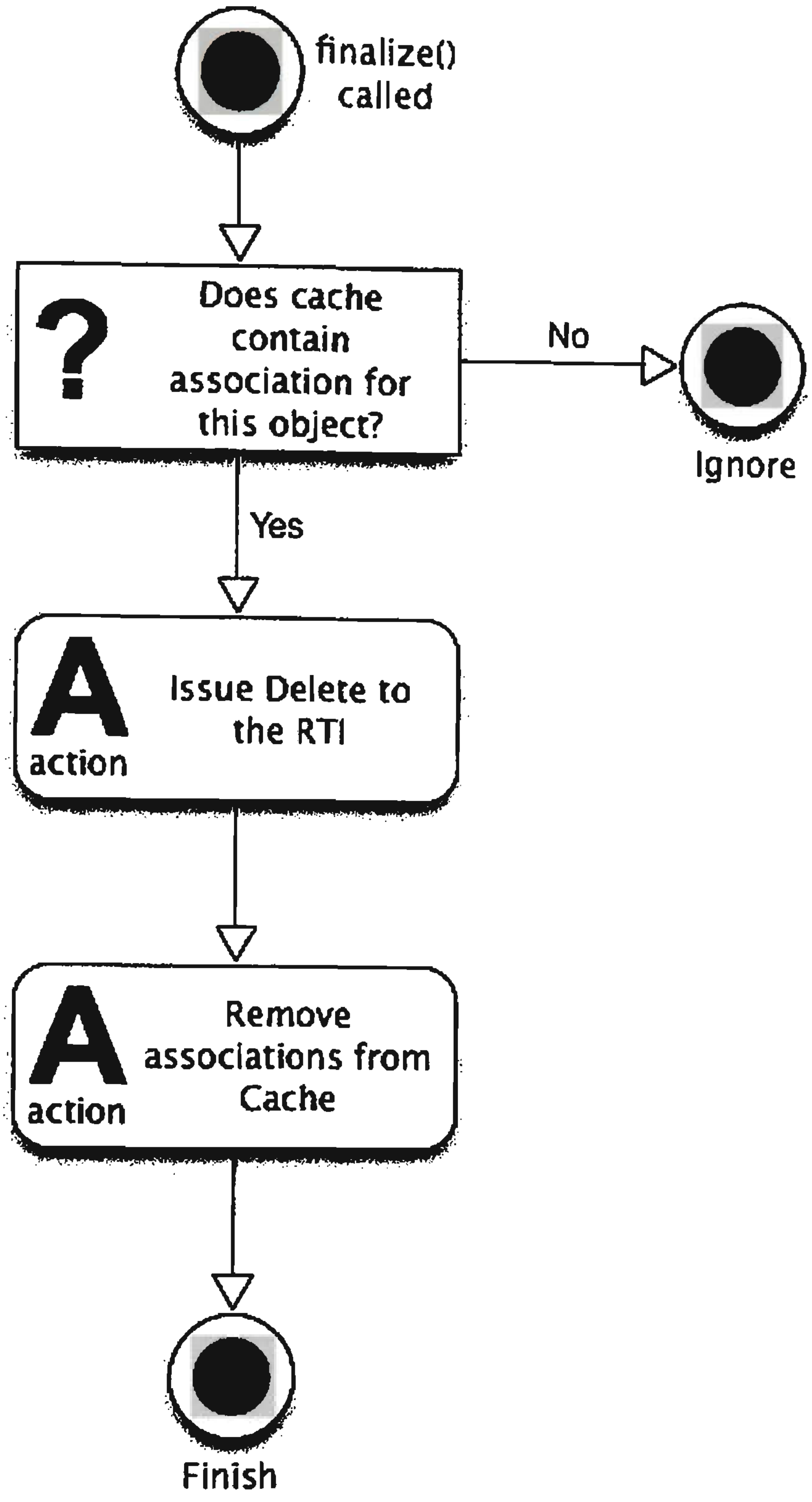

finalize() called
Does cache contain association for this object?
No
Ignore
Yes
action
Issue Delete to the RTI
action
Remove associations from Cache
Finish

Figure 6-6: Object Removed Flowchart

Here, the Cache is consulted to see if the particular instance being removed has an associated HLA registered instance. If it does not, no action is taken. If it does, the Proxy Federate instructs the RTI to remove the object, and any associations are then removed from the Cache.

Despite the simplicity of this process, it is worth noting that with regard to the reference implementation, its invocation can be quite unreliable. Being memory managed, Java has no notion of implicit removal of objects. Thus, a Java OO-model will never remove instances directly. However, through the `finalize()` method inherited from the `Object` class, an instance can be notified when it is about to be garbage collected. By capturing this event in the Generic Aspect, we can access the closest parallel to object removal. That said, `finalize()` is only called before garbage collection, so the likelihood is that some time will have passed between when the object is no longer used by the model, and when it is actually removed. Object-Oriented languages with explicit memory management (such as C++) provide better facilities with regard to this event. In those situations, the destructor could be captured in the Generic Aspect.

Summary

Throughout this section, discussion has highlighted answers to the research question:

"How can the creation, removal and alteration of data within an OO model be replicated into an active HLA federation?"

The processes employed by the handlers discussed here, combined with the Generic Aspect, façade and runtime explanations provided earlier combine to answer this question. However, some of these answers make inferences about solutions to other questions. For example, the conversation on updating owned attribute data is predicated on the idea that data created by remote federates could find its way into the framework. The next subsection tackles this exact problem.

6.3.2 External Data Introduction

The problems of external data introduction are strikingly similar to those discussed in section 6.3.2. Rotating the perspective, these concerns address what happens with data created in a remote federation rather than that created in the pure model. In section 6.3.2 it was shown that although at a high level the conceptual problem is quite simple (data events are observed and HLA events triggered), there are lower level roadblocks that must be addressed. In the case of the previous section, these related primarily to the filtering of events and the handling of unowned data.

When considering external data, the high level concept is again quite simple. HLA events are observed (as received from the RTI) and they trigger actions in the pure model. However, problems begin to arise when pondering how data created and managed in remote federates can be feed into an object-oriented model that might not be expecting it. How will the pure model know where to find this new information? How will the pure model even know there is information that needs to be found? This section answers those questions, discussing the actions taken by handlers as various HLA events occur.

Publish and Subscribe

The filtering of uninteresting data is a major requirement when handling model events obtained through a Generic Aspect. In the previous section, various tactics were used to ensure that only information of interest to the wider simulation was relayed to the RTI. When contemplating the reverse situation, where information is received from the RTI, publish and subscribe facilities of the HLA perform this task on behalf of the federate.

For the publication and subscription features of the HLA to function, information about the interests of the federate must be relayed to the RTI. This step is performed when the runtime first begins, before the model has been permitted to start execution. In section 6.2.1, the model events captured by the Generic Aspect were introduced. The first among these was the capturing of the `main` method, the first method run when a Java program begins. Figure 6-3 presented earlier shows this process. Once the runtime creation process as outlined in that diagram has finished, the Model Façade is notified that it is time to start via the `onStartup()` call. Figure 6-7 is a sequence diagram that illustrates this:

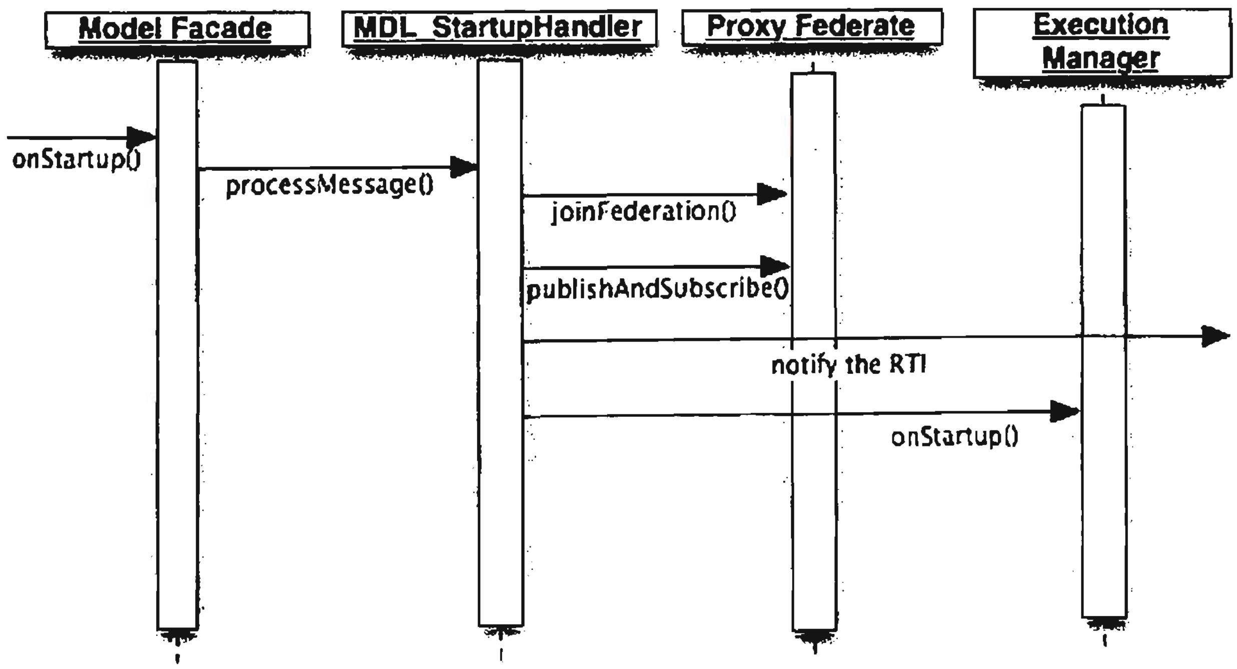


Figure 6-7: Runtime Publication and Subscription

Once the appropriate handler has been notified, it joins the federation and issues all the relevant publication and subscription notices. Having completed this, the Execution

Manager implementation (as defined in the configuration data) is informed that it is time to start up, thus allowing any federate-level agreements covering the synchronization of events to occur.

One problem surrounding this progression is how the runtime determines the actual publication and subscription interests. Once again, this has links back to the disconnect that exists between object-orientation and the HLA. OO has no implicit notion of publication or subscription. There are design patterns that exist to support such an idiom, however, mandating their use seems somewhat onerous. Doing so seems to achieve little more than pushing the development approach of the HLA into the OO realm, regardless of its suitability for the modelling task. The goal of this research is to keep pure models clear of HLA constructs, and such an approach does not fit with this desire.

A preferable approach is to once again turn to the Mapping configuration data. This information describes the overlap in model entities that exist between the pure model and the FOM. As such, it can also serve the dual purpose of highlighting those parts of the FOM that are of interest to the pure model. But should this data be published or subscribed?

The answer deemed most appropriate by this research is **both**. At one level, publication and subscription comes down to permission to register information and permission to receive information respectively. An OO model demonstrates its desire to register instances of a certain type by calling their constructor. Before that, there is no way of knowing, especially in programming languages that have late-binding facilities such as Java, allowing classes to be instantiated without actually making a direct reference to the constructor. With regard to subscription, the same concept applies.

With Mapping data representing all the areas of data overlap that exist between the pure model and the FOM, it seems prudent to issue both publication and subscription announcements all the HLA types referenced by a mapping. Such a methodology means that all situations will always be covered. If there is a link between a pure-model type and a HLA type, there will never be a situation where information is not relayed or received because the publication and subscription inferences were incorrect. In situations where both publication and subscription are not necessary, little is sacrificed through the overstating of interests. This is perhaps the simplest way to link the OO and HLA approaches.

Having presented the approach to publication and subscription prescribed by the Simspect framework, attention can now turn to the actions that occur when various data life cycle related events are received from the RTI. Much like those of section 6.3.2, these fall into three categories: remote object creation, remote attribute updating, and remote object removal.

Remote Object Discovery

Remote object discovery events are triggered when the RTI notifies the Proxy Federate that a new instance has been created. The activities required to deal with these events is minimal. Unlike constructor events for pure-model instances, no determination regarding the interest in the particular piece of information is necessary. As the data provided in the Mappings configuration constitutes the entire subscription set, the proxy will never receive events that should be silently discarded.

Figure 6-8 below outlines the steps used to manage discovery events:

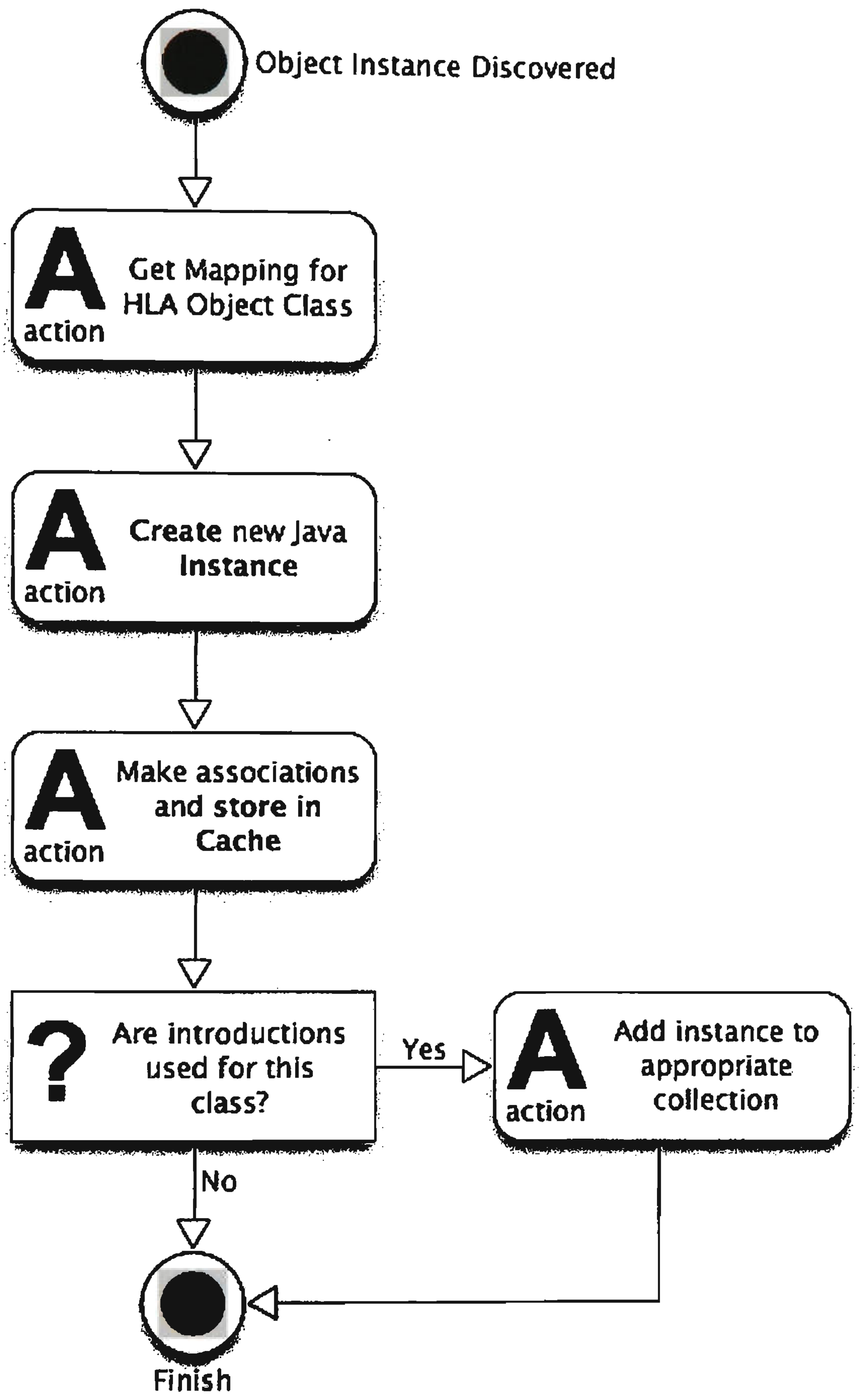

Object Instance Discovered
A action Get Mapping for HLA Object Class
A action Create new Java Instance
A action Make associations and store in Cache
? Are introductions used for this class?
Yes
A action Add instance to appropriate collection
No
Finish

Figure 6-8: Instance Discovery Flowchart

Broadly speaking, this process is the HLA analogue of that invoked to handle constructor events. Mapping information is obtained and used to determine the model class that corresponds to object class of the HLA instance that was discovered. An instance of the model class is then created and associations between it and the HLA instance are made and stored in the Cache for later retrieval.

The only new concept introduced by this diagram is the notion of an *introduction*. To examine why this might be necessary, consider the central problem of the research question these processes address: *How can remote data be introduced into a pure model that is not expecting it?* Once the pure-model equivalent of the HLA instance has been created, how is this information to be made accessible to the model?

Where and How to Introduce Remote Data

Consideration of this problem leads to two basic courses of action, which can rather concisely be summed up as: doing *something* or doing *nothing*. Without access to remote information it becomes impossible for an OO-model to co-operatively perform a distributed simulation. If only local data can be located and operated on, the entire premise of distributed simulation is disregarded. How information can be introduced into a model depends rather heavily on the design of that model.

One common way that many applications access sources of relevant information is through collections. Within some location, specific groups are defined to which entities can be added. At some other time, a process may iterate over the contents of these groups, drawing the necessary information. For example, consider the car race scenario. Given a semantic understanding of the situation, it is simple to identify a Race as a container of Cars. It is reasonable to expect that inside a Race class would be some kind of collection (a list or perhaps a set) in which all the Cars entered in the race are stored. When a discovery event triggers the creation of a new Car representing some remote object, it could be made accessible to the model by placing it inside this collection.

However, this approach has many significant drawbacks. Firstly, if the collection data were private, introductions in this manner would violate encapsulation, a central OO concept. Further, the Race class may implement some kind of mutator method that inserts a Car only after it has passed some checks. Directly inserting information into a collection circumvents this process, potentially breaching the specific rules the mutator enforces.

Building on this problem, the Race class may maintain more than one collection of information about Car instances (perhaps a second collection stores just the driver names). While a custom mutator method could support this by inserting valid instances into the appropriate locations, the sheer volume of alternative approaches for storing information makes direct introduction unattractive. Encapsulation allows these considerations to be hidden. Finally, identifying where information should be stored in an automatic fashion is extremely difficult for all of the reasons mentioned above. While this is not a problem in the context of the first experiment (as configuration data is manually produced), automation is still a major motivation of this research.

Having identified a number of reasons why direct introduction is a brittle solution, the problem of alerting the model to the presence of new data remains. It feels natural or implicit that a solution to this problem necessitates alerting the model to the existence of new data so that it can directly take action. Given this, it seems somewhat contradictory to suggest that the best approach for introducing remote data into a pure model is to totally ignore it. Yet that is precisely the default action this research recommends.

Like many of the methods employed in the Simspect runtime, the root of this approach is an attempt to mimic the typical object-oriented approach. When an instance is first constructed, it is not necessarily assigned to the location where it needs to be. It is only during later processing that the instance is introduced to the location it must reside in. Following the previous example, when a Car instance is first created, it generally would not reside in, or be reference by, the appropriate Race instance. At some point following its construction, it will most likely be introduced to the race via a mutator method.

The important point to note is that these are two separate actions, and following instantiation, no introduction has occurred. Thus, it follows that subsequent to instantiating an appropriate model instance in response to a discovery event, it is not actually necessary to carry out an introduction. This event occurs later, via some model-specific mechanism. As the manner in which this occurs is dependent on the expectations of the model itself, creating something akin to an OO version of a federate level agreement (e.g. Cars must be introduced to a Race via a specific mutator).

In this light, such an approach makes obvious sense. However, it merely defers the problem of data introduction, rather than actually solving it. Depending on the invocation of certain model-specific methods necessitates a mechanism through which those methods can be called. Further, as it is the remote model that has caused this data to come into existence, the responsibility for ensuring that the appropriate sequence of actions executed also resides with it.

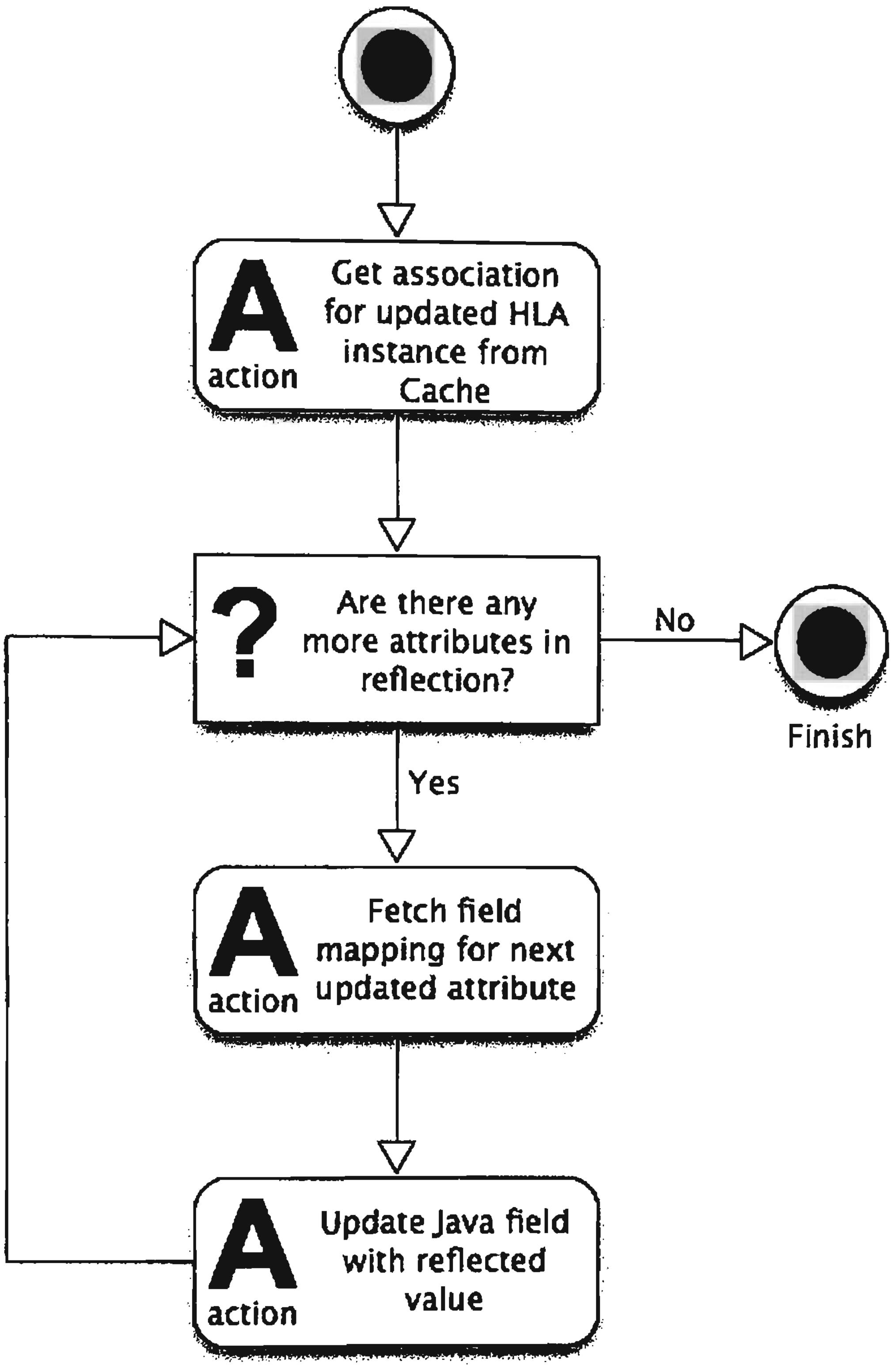
While it lacks the simplicity of a direct introduction, this solution does mimic the way a pure-OO model behaves. To find examples of this, one need look no further than the two test-models used for experimentation. In the car race model, the Race class provides an enterCar() method. In the sushi boat simulation, Customer instances are introduced to a Restaurant through the seatCustomer() and seatVIP() methods, while Dish objects are made available for consumption via the queueDish() method.

Clearly, supporting this style of introduction is not a concern during the handling of remote instance discovery events. The full solution to this problem requires the co-operation of both discovery handling and some facility that allows remote method calls to be identified. The conclusion to this discussion is therefore postponed until section 6.3.3 where the overlap between methods and interactions is presented. In relation to remote data discovery events however, the default action is no action.

Before moving on it is worth mentioning that the reference implementation of the Simspect runtime does support explicit data introductions. Although they were only used in early experimentation (before their limitations were recognised), there are plausible situations where such a facility is appropriate. As stated above, deferring introductions to a later process does create the equivalent of a federate level agreement, and in *certain* situations, the simplicity of automatically introducing newly created data into a collection is both appealing and appropriate. To enable direct introductions, mapping configuration data is annotated with the name of the variable containing the appropriate collection into which new instances should be inserted is all that is required.

Remote Attribute Reflection

Once a local representation of remotely created and managed data has been instantiated (and potentially introduced), some support must exist for ensuring that the values contained within that object are updated. In a typical HLA scenario, attribute reflection services (introduced in 3.2.3.3) are used to convey changes in attribute values. The occurrence of these events can therefore be used by Simspect to update local values. Figure 6-9 outlines the decision process involved in handling such events.



A action
Get association for updated HLA instance from Cache
? Are there any more attributes in reflection?
No
Finish
Yes
A action
Fetch field mapping for next updated attribute
A action
Update Java field with reflected value

Figure 6-9: Attributes Updated Flowchart

The central task of Simspect for a reflection event is to locate the associated model data and update its attributes with the new values. Once instance information has been obtained from the Cache, each of the new values received as part of the attribute reflection is applied to the associated attribute in the local object.

The mapping information for a particular field describes how the incoming data should be interpreted. Data is delivered to the Proxy Federate as an opaque series of bytes. As such, Simspect needs some guidance to interpret the data and transform it into a form that is useful. For example, should the field represent another object instance, the incoming information would be expected to take the form of an integer (as discussed in section 6.3.1). The serialisation information contained in the mapping would inform Simspect that it must first convert the byte data into an integer, and then interpret that integer as the handle of the object that should be applied as the new attribute value. Alternatively, the byte data might represent a string, or a floating-point number. Either way, the mapping contains the information necessary to convert incoming data into useful information.

Although the processes introduced above is quite simple, there is one source of potential problems. In a typical OO setting, attribute data is often declared as being private. This has the effect of restricting access to such an attribute to code within the immediate class. As these values are held in model classes, access to attribute data may not be directly possible. With regard to the reference implementation, this restriction can be subverted.

Java reflection provides facilities to enable and disable accessibility for a given attribute (a facility known as self modification [121]). Accordingly, before any new value is assigned to a private attribute, accessibility is enabled. To ensure that the original intent of the model developer is adhered to, accessibility for the attribute is then reverted to its original status. Increasingly, reflective capabilities such as those employed within the Java environment are becoming more common. Further, efforts to provide reflection-facilitating libraries in languages such as C++ are beginning to become available [25, 64, 96].

In the absence of these facilities, there are other (more complex) approaches that could be used. A Simspect runtime implementation could depend on the existence of setter methods for particular attributes. However, there is no guarantee that these methods would exist, and in the event that they did not, code generation may be employed to generate them. Sadly, this solution is overly complex when compared to the ability to just set the value of an attribute directly. Regardless, this is a problem that further work investigating the application of Simspect methods in non-reflective environments would need to address.

Remote Object Removal

Within the Java language (used for the reference implementation), it is not possible to directly remove an object. An instance will be automatically garbage collected when there are no longer any references to it. When receiving a removal notification from the RTI, there is little that can be done by the Simspect reference implementation to enforce that the associated local instance is also detached.

The main action of the Simspect handler is to delete any association information about the referenced object from the Cache. If no other objects are holding references to the Java instance, it will be garbage collected. In languages that support manual memory management (such as C++) local instance information could be manually deleted when a remove event occurred.

Much discussion in this section has focused on the problems of data introduction. Just as data must be somehow introduced to a model before it can be used, when the RTI instructs the local federate to remove this information, the introduction must be reversed. In situations where a direct introduction facility was used, removal is simple. The collection to which information was added (as defined in the mapping data) is located and instructed to delete its reference to the local instance.

For data introduced via some model-specific series of events, the solution to this problem is the same as for introducing the information in the first place. The task of triggering removal is left to the remote federate. Using the sushi model as an example, when a customer decides to consume a dish, it invokes the `Dish.eat()` method. This in turn removes the dish from availability, causing the `currentDish` attribute of the `Table` class to be updated to `null`. Doing so removes any reference to that particular dish instance[14]. A reference would still be held in the association data stored in the Cache, but once a remove event was received, this link would also be broken, leaving the object eligible for garbage collection. This process is entirely dependent on the expectations of the model itself, but is the approach most consistent with the way OO-models operate.

Summary

Throughout this section, discussion has highlighted answers to the research question:

"How can the creation, removal and alteration of data within an active HLA federation be replicated within a pure OO model that is not expecting it?"

[14] This is not actually the case. When a dish is eaten, a reference to it is stored in another location so that it can be recalled when the Customer attempts to pay for their meal. However, to demonstrate the point, I have ignored this in the example.

The processes employed by the handlers here complement those discussed in the previous section to describe how object and attribute information is created, updated and removed from a pure model. Where the previous section focused on events that were triggered by the pure model itself, this section has examined those that result from actions by remote federates.

A central topic of discussion in this section has been the approaches used for the introduction of remote data. While useful in some circumstances, processes such as the forced introduction of information into predefined collections are brittle, relying on the pure model to take a particular approach for which there is no outstanding convention. On the other hand, relying on the model-specific procedures for data introduction provides a facility that mimics the true actions of a pure model. However, as demonstrated in the examples used throughout this subsection, such an approach depends on the presence of facilities that allow OO-method invocation, something that does not have a direct analogue within the HLA. The next section discusses this problem.

6.3.3 Methods and Interactions

The use of methods (or functions) with object-oriented programming (and programming in general) is central to partitioning work into logical, reasonably sized units. While attribute data represents the state of a particular instance or class, methods provide the behaviour; grouping together related actions into a single, callable unit.

Despite being a fundamental building block of all OO programming languages, the HLA does not provide support for such a feature. Rather, it takes an alternate approach, providing support for something similar to functions, but lacking any sanctioned relationship to a particular object class. HLA interactions are much like functions in the C programming language. Interactions have no association with a particular object class, existing only in a shared, "global" space. Much like the internals of the Objective-C programming language [54], interactions in the HLA provide a facility akin to message passing.

Departing further from traditional programming conventions, HLA interactions can be arranged in an inheritance hierarchy, where parameters from parent types are inherited in child interactions. Even more distinctive is the optional nature of parameters. In traditional programming convention, all parameters of a function must be provided[15]; any parameter can be omitted when sending an interaction.

Is Method Support Needed in Simspect?

Given the reasons above, the ability of interactions to behave "prima facie" as an equivalent for object-oriented methods does not exist. Nonetheless, considering the extremely important role methods play within OO, some parallel feature is required.

In a broad context, one could argue that the last statement is a falsehood, and that the only lasting or important result of any method invocation will be the changes it makes to data shared among a federation. According to such an argument, there would be no need for a method support, as the only measurable results would be attribute reflections for any relevant changed values. This is certainly true to an extent, however, it does miss the general point of using functions in the first place.

Functions are a grouping facility. While it would be possible to contain all the logic for a particular model within a single, sizeable, section of code, functions allow this task to be broken down. Apart from being a primitive feature to partition an action into a set of smaller components, methods also centralise behaviour, removing redundant code. Removing methods from consideration in favour of focusing solely in changes in attribute data would force each federate to understand how other federates intend to manipulate their attributes. A federate that controls a car race would lose its ability to request that all cars update their `distanceTravelled` at regular intervals as there would be no mechanism to pass this message.

Beyond grouping considerations, methods also provide polymorphism. By grouping any changes behind a function, different `Car` implementations can respond to the same message in different ways, altering related attributes according to different algorithms. In the context of the HLA, this means that one particular federate could contain an algorithm for calculating the distance travelled in a period of time that is entirely different from that provided by another federate. Ignoring the message-passing role of functions would remove this ability.

The use of method-like facilities plays such a central role in the development of OO-models that to ignore it would necessitate mainstream developers alter their entire approach to structuring and developing them. There is a clear need for *some* harmony to

[15] Some languages provide support for "default" parameter values. In these situations, parameters can be omitted, and the default value is substituted. However, the HLA does not support this feature.

exists between OO methods and HLA interactions. Although there are significant differences, at a fundamental level they can be tasked to serve the same purpose. This leads to the research question driving the discussion in this subsection:

"How do object-oriented methods translate to HLA interactions?"

Mapping Methods to Interactions

Finding common ground between methods and HLA interactions is a considerably deeper task than it may first appear to be. If invocations of a particular method are to be communicated to remote federates via interactions, a modicum of conceptual synergy must exist between the behavioural model of that method and of interactions in general. The behaviour and usage scenarios of interactions mean that there are limits on the types of functions for which a reasonable translation can be made. Put another way: not all methods logically map to interactions.

Return Types

Object-Oriented programming language functions have a return type. For situations when there is to be no return type (perhaps on a request to perform some action, such as the `moveCar()` method), a function can signal the absence of any return type by declaring it to be *void*. Interactions are transient, asynchronous messages for which no ability to return a value exists. Even if such a provision were provided, the multiplexing nature of HLA message transmission (any number of federates can subscribe to an interaction) means that determining the single valid source of return information would present logical challenges.

Given this, it is quite clear that no genuine overlap can exist between methods that require a return value, and any interaction-based representation of them. Although some request/ response mechanism could be constructed to subvert these problems, such an approach would be seem unnatural, and would constitute too large a misrepresentation of the purpose of HLA interactions.

Static Methods

Within object-oriented parlance, methods can have differing scopes. *Instance* methods are those that are invoked directly on a single instance of a particular class. *Static* methods are those associated with the class in general, rather than any specific instance.

As discussed later, relatively straightforward mechanisms can be constructed to associate a given interaction with a specific "target" object instance. However, as the HLA purposefully omits any link between data and behaviour, there is no class with which a "static interaction" could be associated. Any mechanism to address this shortcoming would once again be far too misrepresentative of the HLA model to constitute a valid link between it and OO. Thus, static methods can also be ruled out as potential candidates for communication via interactions.

Mutator and Accessor Methods

One final class of method to consider is accessors and mutators. As accessors (or "get" methods) require a return value, they can be immediately identified as unsuitable candidates for interaction representation. Mutator methods however fit the general profile of "interaction friendly" method forms in that they have no return type and they are not static.

However, some consideration must be given to the purpose of mutator methods. Broadly speaking, they exist to provide an avenue through which external types can request alterations to the value of attribute data (some of which may be private). In a HLA setting, where the responsibility for changing attribute data is seen to be the sole responsibility of the owning federate, such an action generally has no place. Further, within the Simspect architecture, should any call to a mutator method actually affect some change, this would be handled via the field modification processes. Therefore, despite taking a form that can be represented as an interaction, the use case surrounding mutator methods means that there is little real value to representing them as interactions.

Public and Private Methods

Object orientation introduces the concept of visibility. Each method has an associated identifier that conveys from whence it may be called. Although they tend to differ slightly between OO-implementations, visibility specification takes one of three forms: public, private and protected. Unless a method is declared as public, it can only be invoked from code that exists either within the same class, or within a subclass. The public interface of a class does not include private methods. Rather, private methods are deemed to be internal processing, relevant only to the particular class.

If it for this reason that private methods cannot be considered valid candidates for interaction representation. The manner in which a particular class functions internally is of no interest to external entities. In OO, these external entities are other classes (or instances of said classes). In the HLA, these external entities are other federates. The intermediate behaviour of a particular federate as it executes its desired algorithm is of no interest to a federation. Rather, the results of that manipulation form the lasting interest. Thus, as private methods are internals working, the need not be represented as interactions.

On the other hand, methods that have public visibility signal that they can be called from anywhere, and as such, linking these methods to interactions so that they can be remotely invoked is logical. Just as external classes can invoke public methods within an OO-model, external federates should be permitted to invoke public methods for object instances managed remotely.

Representing Methods as Interactions

The discussion above has outlined the general form a method must take to be considered a reasonable candidate for representation and transmission as an interaction. To summarise, within the Simspect framework, candidate methods may only have a valid interaction representations if they:

- Have **public** visibility
- Do **not** have **static** scope
- Have a **void return** type

Having established the requirements for a method to be represented as an interaction, there are still a number of questions that remain.

Association with Object Data

A major difference between methods and interactions is that the latter are never directly associated with any object data. OO instance methods however are invoked on a specific object instance. To accurately represent this behaviour there must be some mechanism to identify the particular object to which an interaction representing a method call is associated. That is to say, the target of the method must be identified.

To achieve this, Simspect mandates that within an HLA object model, any interactions that are to represent method invocations should be declared in a specific hierarchy. The snippet below provides an example of this:

```
(class InteractionRoot reliable receive
     (class MethodCall reliable receive
       (parameter targetObject)
       (class Race_enterCar reliable receive
         (parameter car)
       )
)
```

Listing 6-5: Method Call Interaction Hierarchy

This is the mandated hierarchy for those interactions that are used to represent method calls. Each interaction must extend the `MethodCall` interaction class, and as such, inherits the `targetObject` parameter. This parameter is used to associate a specific interaction with the object instance to which its invocation refers. The value of the parameter should be the HLA object handle of the desired instance.

Given that any federate can subscribe to any interaction it desires, there will be situations where a method call interaction is received by a federate that does not manage the target object instance. In those situations, only the federate that manages the object in question is expected to take any action. While other federates can observe, no action should occur as a result of this interaction.

In the example above, should the `enterCar(Car)` method be called, subscribing federates would receive an interaction with two parameters. The `targetObject` parameter would be the object handle of the `Race` instance on which the method was called, and the `car` parameter would be the object handle of the `Car` instance that is being entered into it[16]. This approach to making associations between data and functions is not new and has been used in many situations for many years. A primary example of this is the Objective-C programming language that uses a similar process to associate C-style functions with particular sets of data [54] via a process they call "message passing."

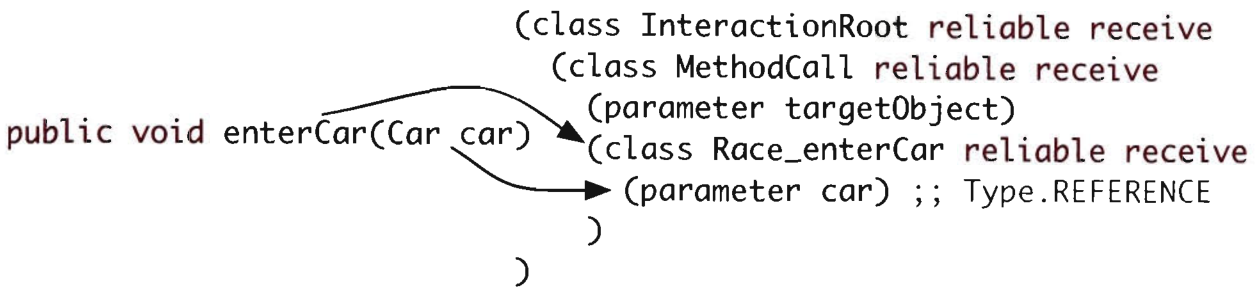
<u>Associating Methods and Interactions</u>

Under the OO methodology, a unique method is identified through its signature, and the class in which is resides. Interactions however reside in their own hierarchy, separate from any object class. To successfully map between a specific method and the FOM name given to the interaction representing it, additional information is required. Continuing the

[16] As discussed in section 6.3.1, object references are serialized into the HLA object ids for the registered instances representing the object.

example from above, there must be some way to associate the enterCar() method with the Race_enterCar interaction.

This information comes in the form of mapping configuration data. Just as mapping information defines the interesting elements of a pure model, and how they relate to a specific HLA object model, the same data identifies those methods of interest within the model, and the interactions to which they are mapped. Figure 6-10 shows how method information is mapped to interaction data:



```
                          (class InteractionRoot reliable receive
                            (class MethodCall reliable receive
                              (parameter targetObject)
public void enterCar(Car car) (class Race_enterCar reliable receive
                                (parameter car) ;; Type.REFERENCE
                              )
                          )
```

Figure 6-10: Mapping Methods to Interactions

This figure is a visual representation of the data contained within the mapping. Within the reference implementation, the XML mapping data states that the enterCar() method is associated with the Race_enterCar interaction, and that the first parameter to the Java method is associated with the car parameter in the interaction[17]. In this situation it is important to note that the choice of interaction name is entirely arbitrary and the idiom of prefixing the class name to the beginning of the interaction name is purely for illustrative convenience.

Mapping information for a method/interaction association also must inform the runtime of the parameter *type*. For example, consider the following XML information that is taken from configuration data used in the reference implementation:

```
<method jmethod="testcode.racesim.Race.enterCar"
        hmethod="InteractionRoot.MethodCall.Race_enterCar">
    <param jposition="0" hname="car" type="REFERENCE"/>
</method>
```

Listing 6-6: Method Mapping Definition

[17] Java reflection does not support identifying parameters by name, only by position.

In this snippet it is possible to see how the first Java parameter is associated with the HLA model parameter "car". The type information for the parameter is also specified, providing the runtime with the necessary information to convert the value passed to any java method execution into an appropriate form (and back). See section 6.3.1 for more information on how values are converted.

Method Events Within a Model

During the execution of any pure model, a large number of method execution events will occur and be passed to the runtime for handling. Figure 6-11 below outlines the processing that occurs when an event must be handled.

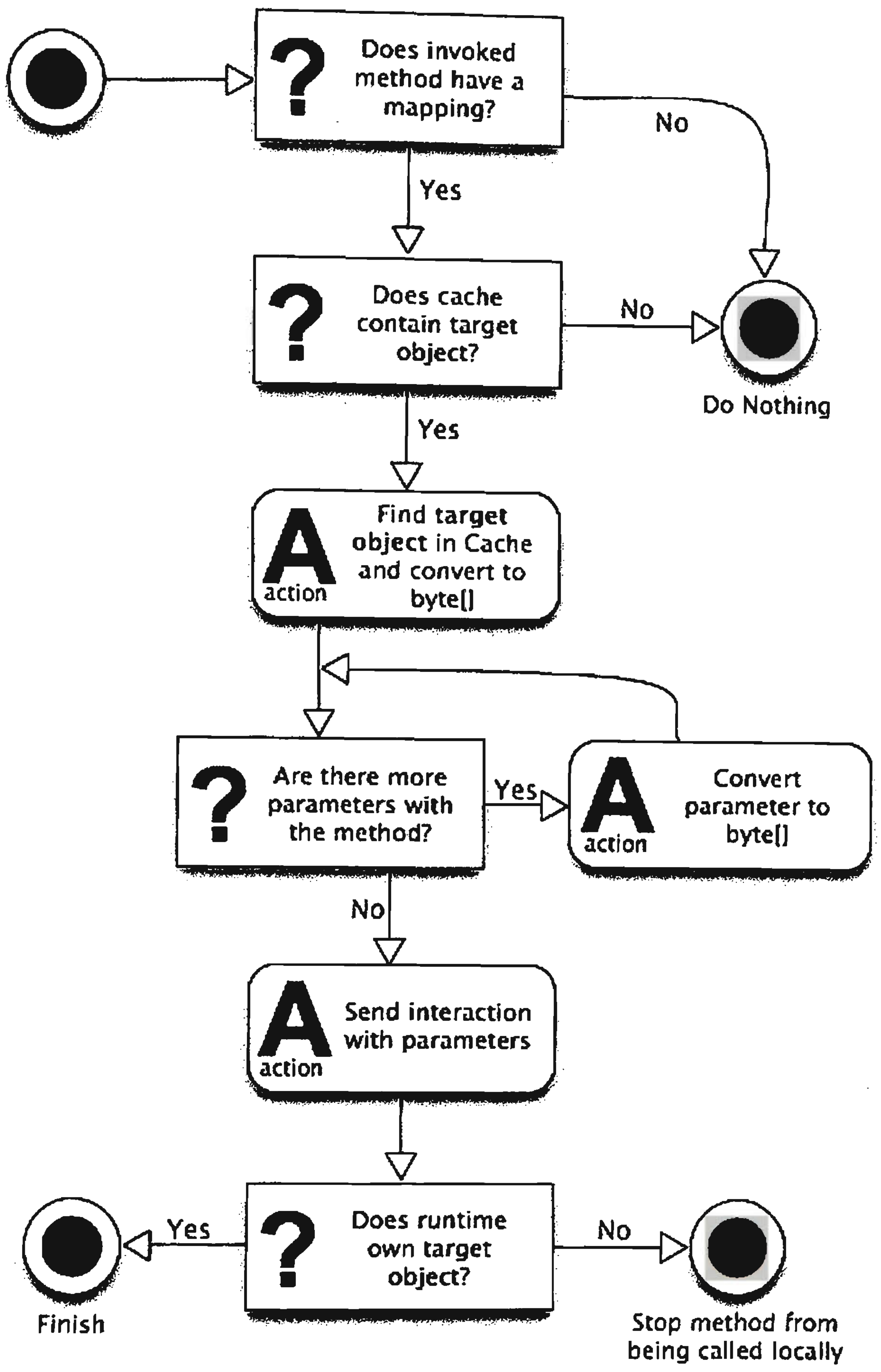

Does invoked method have a mapping?
No
Yes
Does cache contain target object?
No
Do Nothing
Yes
action
Find target object in Cache and convert to byte[]
Are there more parameters with the method?
Yes
action
Convert parameter to byte[]
No
action
Send interaction with parameters
Yes
Does runtime own target object?
No
Finish
Stop method from being called locally

Figure 6-11: Method Execution Flowchart

As with object data, not all of these events will be of interest to the wider federation. The first step in the decision process outlined in figure 6-11 is to filter out any methods that should be ignored. Before moving on, a brief discussion on the three-stage process to method filtering is necessary.

Filtering Potential Methods

As with object data, not all methods will be of interest to the broader federation, and as such, some filtering is required. The first step to filtering out unwanted method invocations is to ensure only those of a valid form are captured. This occurs within the generic AOP Aspect. Consider the following snippet:

```
1      /** pointcut to capture all valid methods */
2      protected pointcut methodCall( Object target ) :
3            execution( public !static void *.*(..) ) &&
4            !execution( * *.get*() ) &&
5            !execution( * *.set*(..) ) &&
6            ignoreList() &&
7            target( target );
```

Listing 6-7: Method Capturing Point Cut

This pointcut will only capture the execution of methods that take a valid form. Line 3 mandates that a method must be declared as public, non-static and have a void return type to be captured. Lines 4 and 5 rule out any methods beginning with the "get" and "set" idioms used to represent accessor and mutator methods[18]. This pointcut forms a kind of "first pass filtering."

When the Generic Aspect captures a method execution, information of the event is passed to the Model Façade, and then on to the appropriate handler. It is at this point that the second level of filtering must be applied. As shown in figure 6-11, if there is no mapping information for the method that is about to be executed, Simspect will ignore the event, and no interaction will be sent. This filters out validly formed, yet uninteresting calls (as interesting methods will have mapping data).

The final step in the decision process is to locate the target object on which the method has been called. If the method has **not** been called on data for which HLA information exists (either registered by the proxy, or created remotely), there is little point notifying the

[18] In programming languages that have slightly different conventions for representing get/set methods, these lines would be modified to suit that convention.

federation. In such cases, the method invocation is deemed internal processing, and no action is taken.

Parameter Serialization

Once the mapping data for a particular method has been located, the parameter values that are to be sent with the interaction should be created. Each parameter provided with the method execution in the pure model is converted into a form acceptable to the HLA using the mapping information. When no more parameter information exists, an interaction containing all the created data is sent to the RTI.

Method Invocation and Unowned Data

Once an interaction has been sent, the final step required is to determine whether or not the method should be allowed to execute locally. As with field modification (section 6.3.2), method execution events are triggered via *around* advice, and thus, the actual execution of a method can be stopped from occurring at all. Whether or not this should happen depends on who created and owns the object instance in question.

If the instance was created locally, the pure model is responsible for managing it. As such, the method *must* be allowed to execute. To prohibit it from executing could introduce errors by removing the ability of the model to manage its own data. However, if the data was created remotely, some other federate is responsible for managing it. In such a case, the method is prohibited from executing locally.

This approach is necessary for a number of reasons. Firstly, just as a model must be permitted complete control over local data for consistency reasons, it must also be restricted from potentially altering data it is not responsible for. The actions taken by the local model in response to a particular method may differ from those of the federate responsible for the data. Allowing the method to execute could impose an invalid algorithm, altering field values and making them inconsistent with the rest of a federation. By restricting method execution on unowned data, proper polymorphism is enabled.

For example, consider the flow of events that occur in response to the `moveCar(double)` method being invoked on a `Car` instance. If the `Car` is a locally controlled instance, it makes logical sense to execute the model method and to let the pure model update the instance according to the algorithm it implements. However, if the `Car` is modelled by a remote federate, the strategy used to determine how far it has moved may differ from that used by the local model. Allowing the method to execute locally would alter the values according to the algorithm implemented in the local model, not the algorithm imposed by the federate responsible for that particular instance.

By making a decision on whether or not the local method execution should be allowed to proceed, potential errors in a distributed model are avoided, and proper polymorphism is enabled. Further, exploiting this particular approach also helps when attempting to build a pure, monolithic OO model that defers certain calculations to remote participants (even though it has no notion of application distribution). This particular characteristic is discussed in chapter 9 when talking about authoring monolithic models to behave in distributed simulations.

Interaction Events Received from HLA

Having seen how model method execution events are communicated to the HLA via sending of interactions, it is now time to turn attention to how the Simspect runtime handles such interactions when the Proxy Federate receives them.

Capturing method execution events within the pure model allows Simspect to notify other federates of that action. In the case of unowned data, this is vitally important, and the interaction performs a role similar to a very loosely coupled remote procedure call. The Simspect runtime must also consider the reversal of this situation, where the Proxy Federate receives an interaction representing a method invocation. If such an interaction relates to a target object that is controlled locally, the runtime must be able to somehow invoke the appropriate method within the pure model.

The provision of such a facility provides a rudimentary form of remote method invocation, allowing OO semantics to be accurately represented within the HLA. Through this capability, public methods can be invoked by remote federates. If the receiving entity is an actual HLA federate, these interactions can be called to ensure a pure model behaves in the proper manner. If the remote federate is another Simspect-based model, these interactions are converted into method calls. How the local model handles such requests (if indeed it handles them at all) is an entirely separate matter.

As with other Federate Ambassador call-back methods processed by Simspect, there is no need for the explicit filtering that takes place when model events occur. Section 6.3.2 laid out how publication and subscription is handled when the Simspect runtime starts up. Just as any object data for which mapping information exists is both published and subscribed to, any interaction information is given the same treatment. Given this, only the method related interactions that Simspect has been defined to have an interest in would ever be received.

With the RTI performing the bulk of the necessary filtering on behalf of the model, the event processor that executes when an interaction is received can ignore such tasks. Figure 6-12 outlines the process taken by the interaction event processor:

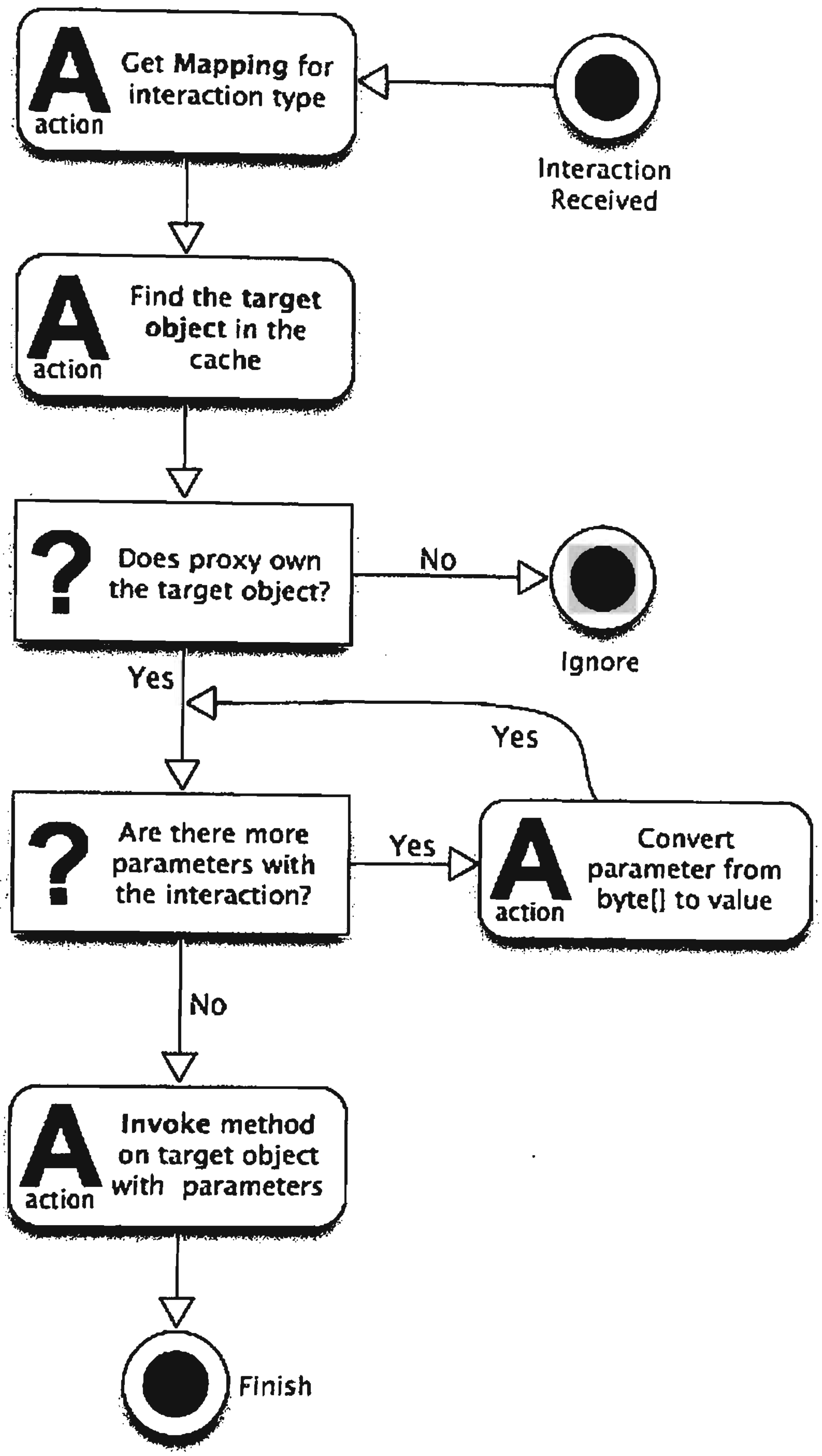

Get Mapping for interaction type
action
Interaction Received
Find the target object in the cache
action
Does proxy own the target object?
No
Ignore
Yes
Yes
Are there more parameters with the interaction?
Yes
Convert parameter from byte[] to value
action
No
Invoke method on target object with parameters
action
Finish

Figure 6-12: Interaction Received Flowchart

The first step is to locate the appropriate mapping information for the interaction. This data is guaranteed to exist, as if it did not, the interaction would never have been subscribed to in the first place. Having located the mapping information, the target object is sought from the Cache. Due to the multiplexed nature of interaction sending in the HLA, it is probable that over its lifetime, a number of interactions relating to unowned data will be received. In these cases, the request is immediately ignored.

If the target object represents a locally created and managed instance, the runtime is made responsible for invoking the associated method within the pure model. Once all the received parameters have been converted into an appropriate form (as discussed in section 6.3.1), the model method identified in the mapping data is invoked on the target object, and the process is complete.

Summary

Just enough common ground exists between method invocations and interactions to allow a palatable compromise to be found between the two. OO-models depend heavily on methods to group behaviour and act as a request and communication facility. While interactions fulfil a similar role within the HLA, their asynchronous nature means that they can rarely be considered a vehicle through which guaranteed remote procedure calls can be made.

The hybrid approach proposed in this section blends the core values of both approaches. Method calls can be made across model boundaries, supporting polymorphism in a shared modelling environment. At the same time, pure HLA federates can trigger behaviour within an OO-model, while responding to or ignoring the method invocation semantics of those models when requests flow in the opposite direction.

However, one quandary that is evident throughout this entire section is the enhanced role federate-level agreements play (although, in this context, it is perhaps more accurate to refer to them as model-level agreements). The approach presented here can give rise to situations where a model has expectations of a remote federate that are never fully met (or vice versa).

For example, the model controlling a particular `Race` would expect that all federates would respond to invocations of `moveCar()` by advancing the distance travelled for a particular `Car` instance. If the instance is managed by another Simspect controlled model, there is no concern that the method will be invoked through the processed outlined above. However, it is perfectly valid behaviour for a plain federate to ignore the representative interaction altogether, thus causing some cars to never move (and a race to never end). Although this likelihood of this particular scenario may be small, it does demonstrate that if the expectations of a particular model or federate are betrayed, errors can potentially arise.

Despite this, Simspect does provide some consolation. By providing a methodology that allows models to be developed free from the HLA, rectifying such semantic misalignments can be achieved entirely via pure-OO code, thus reducing the time and effort required to solve such problems.

6.3.4 Ownership Management

The topic of ownership management cannot be considered in isolation of other concerns relating the operation of the Simspect runtime. It is intimately linked to problems like data management (6.3.2) and method/interaction crossover (6.3.3). As such, the answer to the research question presented at the beginning of this chapter has been split up and addressed during discussion in the relevant areas.

That said, the general theme arising from these discussions is that the best approach to reconciling the monolithic world-view taken by a pure model with the shared approach of the HLA is to ignore any requests for processing data that is not managed locally. When methods are invoked on data for which a remote federate is responsible, it does not proceed within the model itself. When the proxy attempts to directly manipulate an attribute belong to an object modelled externally, that request is also stopped from proceeding.

External data is the responsibility of external federates, and allowing any local modification that is not the result of an explicit instruction (such as an attribute reflection) serves only to introduce potential errors and inconsistencies in the representation of that data across a federation. The major benefit of this methodology is that it enables true shared modelling to occur, and polymorphism to be implemented across a federation.

6.3.5 Logical Time

The concept of logical time is used within the HLA to guarantee the ordering of events and provide some level of synchronization in execution among multiple federates. Time management and advancement is a complex topic, and one that could easily warrant considerable discussion in its own right. Determining the best way to generically support the many potential mechanisms for representing time within a pure model is a topic on which an entire study could focus alone. However, the primary goal of this research is to define a broad level of solutions in the pursuit of providing an environment in which object-oriented models can function as distributed simulation components. As such, an in-depth consideration of time management is well beyond the scope of this work.

That said, time management is a central concept within many HLA simulations, and as such, this work must present at least a rudimentary solution for synchronizing a representation of time between a monolithic pure model and a HLA federation.

Representing Time

To this point in time, the HLA has refrained from demanding that time be represented in any particular manner. Abstract types are used to represent time, allowing custom representations to exist depending on the requirements of the federation. This approach, while valuable, has also often been recognised as a significant contributor to the lack of federate portability at an API level [107].

Although it allows absolute flexibility, leaving decisions on how to represent time within a simulation up to the developers introduces many problems, and as such, the Simspect runtime must introduce some restrictions.

Firstly, time must be represented by a single, static variable in some class accessible within the model. The visibility of the variable does not matter. The demand that it must be static, and not an instance level variable ensures that there is only one representation of the value of time. If instance variables were allowed, some way to tell which instance contained the appropriate value would be necessary.

Secondly, the variable must be of the primitive type **double**. This is not actually a requirement mandated by Simspect, but rather by the reference implementation. Other implementations could use an entirely different data type, but the important point is that the type is known. This allows the runtime to make determinations on how to manipulate the value and reconcile the model representation with that required by the HLA (by serialising it to and from the appropriate form).

To enable time management, the runtime must be told which static variable represents time through the configuration data it is given at start-up. By default, this information is omitted and time management is ignored. However, if it exists, any Proxy Messages will be sent with an associated timestamp. Further extensions to this work could investigate ways for allowing finer levels of control over which data is subject to time-stamped updates and which is not.

Managing Time Advancement

Having established how time should be defined within a model, attention must be given to how the advancement of time can be constrained so as to keep it in step with the rest of the federation. With knowledge of the representation of time, this process is actually quite simple.

Casting consideration back to figure 6-5 (in section 6.3.2), there was a provision within the field modification handler for dealing with any attempted alterations to the variable that represents time. When such an event is captured within the model, the runtime takes an alternate path to that normally used to handle field alterations.

The first step is to request time advancement from the RTI to the new value a model is attempting to assign to the static variable representing time. Following this request, the event handler stalls execution. It sits in a tight loop, continually ticking until the Proxy Federate is notified of that the advance has been granted. At this point, the model is allowed to continue executing, and the field modification is permitted to occur. Any time the model attempts to change the variable representing time, it is forced to halt until the RTI permits it to move forward.

The benefits of this approach are clear. There is no need to alter the model in any way in an attempt to enable the distributed management of time. The model does not know that it is being halted in order to meet the distributed time requirements, or that it is being halted at all. The process transparently introduces HLA-based time advancement into pure-OO models.

The solution presented here will work for any model that conforms to the desired scenario (where time is represented as a single, static variable). However, it does only provide a solution for a method of time advancement known as "time stepping." The HLA itself supports other forms (such as event-based and optimistic time advancement) for which this method provides no solution. As stated at the beginning of this section, the topic of time management in the HLA is a complex and sizeable topic. The simplistic approach presented here, although not ideal, will work in a number of common situations. Any consideration for a more robust approach is a topic for further work.

6.3.6 Federate Level Agreements

The problems of federate level agreements have been highlighted throughout this chapter, and in attempting to blend OO and HLA style approaches, Simspect itself introduces restrictions that can fall under this banner. The simple fact of the situation is that providing automated support for the individual expectations of particular federates in a generic fashion is just not possible. The sheer number of services that can be broken as a result of different usage patterns puts this goal beyond reach. The situation gets worse when considering the potential combinations of those services that could be embodied in the expectations of a particular federate.

These problems are not specific to the HLA, but rather, to software development in general. In any situation where a model makes use of a particular API, it is creating a dependence on that interface and demanding its existence in order to operate correctly. Earlier subsections pointed out that although many of the methods proposed in this chapter help bridge the usage gap that exists between monolithic OO models and HLA based distributed simulations, these same approaches introduce new areas for potential semantic misalignment. The experimental car race model expects remote federates to behaving in specific ways, responding correctly to interactions that ask them to advance instances of the `Car` class that they are responsible for. If a remote federate cannot fulfil this obligation, execution will break down and error will occur.

Despite all this, the use of Simspect does provide some significant advantages for anyone wishing to address these concerns. The Simspect methodology allows pure object-oriented code to execute a model co-operatively with other federates. Without the need to consider low-level HLA details, the amount of effort required to alter a model so that it conforms to the expected behaviour is drastically reduced.

Further, a solution for dealing with the execution requirements of a legacy HLA simulation is provided through the Execution Manager. Although the development of such a component would demand HLA knowledge, it is only necessary in situations where an existing HLA simulation must be dealt with. In those circumstances, it seems entirely reasonable to expect that *some* HLA expertise would be available. By providing a strong framework supporting the development of an execution manager, the burden of producing a component for such specialised, fringe situations is reduced further.

Although it is beyond the reach of this research to support entirely arbitrary federate level agreements in a generic manner, the approaches discussed in this chapter have outlined how the burden of addressing such demands can be minimised.

6.3.7 Ticking and Call-back Invocation

One facet of the HLA that has yet to be discussed anywhere in this chapter is the notion of "ticking". It is the responsibility of all HLA federates to inform the RTI when they are ready to receive Federate Ambassador call-back notifications. This process is commonly invoked via the `RTIambassador` classes `tick()` method[19]. If call-backs are not periodically processed, the federate can become starved, and in turn, adversely affect the behaviour of other federates within the federation.

This approach only becomes a problem when one considers that a pure model has no notion of application distribution, or that it needs to process incoming call-back messages. Identifying points within a model that represent natural "breaks" where notifications could be solicited is difficult to achieve in any generic fashion. However, the methodology proposed by Simspect does provide a rather simple solution to this concern.

As discussed in the previous sections, the generic nature of the AOP Aspect means that many events that hold no interest for the HLA are captured. Although these events are quickly identified and filtered out by the appropriate event handlers, their volume is considerable. As such, each of these notifications represents a perfect opportunity to insert calls to the `tick()` method.

Inside the advice provided by the Generic Aspect, calls to `tick()` are executed before any event information is passed on to the Model Façade. The sheer number of these events means that the probability of starving the federate of call-back notifications is highly unlikely. On the one hand, the considerable quantity of events is beneficial, removing any concern for proxy starvation. However, in high volume situations, it can also become a burden, with solicitation of call-backs occurring too often. Although the reference implementation follows the basic recommendation presented here, a more intelligent approach could easily be applied (perhaps only invoking `tick()` once a timeout has occurred since the last invocation). Regardless of the precise approach taken, by utilising this by-product of the Generic Aspect, there is no need to delve into complex solutions aimed at identifying natural pauses within a model where this behaviour can be inserted.

6.3.8 Summary

This completes discussion of the methods proposed by the Simspect runtime framework. Throughout this section, the research questions identified at the beginning of this chapter have been addressed, discussing the problems that arise and the proposed solutions. Through a combination of AOP and the approaches presented here, a pure-OO model,

19 Although the IEEE 1516 specification altered the name of this method, the general approach remained the same.

with no notion of application distribution, can co-operate with and participate in an HLA-based distributed simulation.

Although the contribution of this research goes well beyond this new ability (as discussed in subsequent chapters), the only major task that remains in relation to the runtime framework itself is to validate that it works. Experiment One is designed specifically to demonstrate this, taking two object-oriented models and placing them into a distributed simulation where they complete the shared modelling of their respective scenarios.

6.4 Experiment One

Before an OO model is ready for use with the generic HLA Aspect introduced in the previous section, a number of tasks must be undertaken. These include the creation of deployment artefacts (such as object models), and the specification of weaving rules that describe how the model and Aspect are combined. As referenced earlier, in a typical AOP environment, these are all manual tasks. The purpose of this experiment is to validate that generic AOP Aspect and demonstrate how model and infrastructure code can be created separately. Although the manual processes of creating the deployment artefacts requires specialist HLA knowledge, this is acceptable in this case. Later experiments will show how to automate these parts.

Before the experiment can proceed, two documents need to be manually produced. Firstly, a file describing the object model (FOM) for the federation needs to be created. The contents and structure of this file are intended to be a HLA representation of the object model used in each of the pure-OO models. Secondly, a Simspect configuration file needs to be written. The most important part of this file is the mappings data that describes how Simspect can convert data between its OO form and the HLA form demanded by the FOM.

6.4.1 Experimental Results

The criteria for assessing the results of this experiment were outlined in the table in section 5.2.3. To help show that the solutions discussed throughout this chapter successfully meet these requirements, the following items must be gathered and inspected:

- The pure model code
- The output produced by each model in its pure form
- The output produced by each model after it has had Simspect woven into it
- The output of the companion federate

The results for each of these criteria will now be individually addressed. For reasons of brevity, the log files that were captured during experimentation are not provided in the appendices. Some of them are quite large (greater than 15,000 lines) and rather than include them in the text, they are provided as part of a supplementary package that accompanies this work.

Criterion One: AOP-model must remain HLA free

A visual inspection of the code for each pure-OO model can quickly confirm that they are free of any HLA considerations. To this end, the code for each model included in the supplementary package that accompanies this work.

Criterion Two: AOP-model must execute without error

The AOP-model is the name given to the pure-model once it has been attached to the Simspect environment. To ensure that Simspect does not cause any runtime errors, the AOP-model must be able to run to completion. This can be verified by inspecting the Simspect Runtime log for each AOP-model. Earlier in this chapter it was noted that the Simspect Aspect used an "around advice" with regard to capturing the main method. This type of advice allows the Aspect to wrap around the execution of a method and determine whether that method should execute or not.

The code from inside the Simspect Aspect for this advice looks like this:

```
// proceed and execute the model main method
logger.debug( "{BEFORE} main" );
proceed();
logger.debug( "{AFTER} main" );
```

Listing 6-8: Main Method Advice

This listing shows that if the model were to execute successfully, there should be no errors in the output, and a log entry should be made after the completion of the method. Inspecting the log files for each simulation, we can locate the following lines, indicating success:

```
18201 DEBUG [main] simspect.runtime: {AFTER} main
18202 DEBUG [main] simspect.runtime: facade.onShutdown(): stopping runtime
18203 DEBUG [main] simspect.runtime: facade.onShutdown(): stopped runtime
```

Listing 6-9: Main Method Capture Log

Criterion Three and Four: Object Creation

These criteria relate to the handling of object creation in both the AOP-model and the federation. To successfully meet these requirements, the federation needs to be informed of all interesting data created inside the AOP-model, and the AOP model needs to create local copies of all interesting data[20] created in remote federates.

To demonstrate this, the log files for the AOP-model and the companion federate are inspected. When data is created inside the AOP-model, the constructor is captured and an object registration is sent out to the federation. The listing below comes from the AOP-model's Simspect log file:

```
DEBUG [main] simspect: {CNSTR} Instance created [testcode.racesim.Car] (hash:
16453941): HLA instance registered and cached, handle: 8
```

Listing 6-10: HLA Object Registration Log

Here, an instance of the Java object `testcode.racesim.Car` is being created and a corresponding HLA instance is registered, the handle for which is 5. The hash value uniquely identifies the Java instance and can be used when looking at later log entries. In the companion federate (known as the "racewriter" federate for the race simulation), the log shows the discovery of this instance:

```
INFO  [main] racewriter: {DISCOVER} Instance of [ObjectRoot.Car] with handle [8]
```

Listing 6-11: HLA Object Discovery Log

In the reverse situation, Simspect needs to create a local proxy of any data that is created in a remote federate. Additionally, the final results of the model execution should reflect the presence of this remotely created data (this is shown in criteria nine). The two log extracts below show an object (hla handle: 4) being registered by the companion federate, and that object being discovered by Simspect. Simspect then goes on to create a proxy instance of the class `testcode.racesim.Car` for that instance of the HLA object class `ObjectRoot.Car` (as identified by its hash code).

[20] As defined earlier in this chapter, interesting data is data for which mapping information exists in the Simspect configuration. As this configuration is manually created for experiment one, a user (knowledgeable in the HLA) makes this determination.

```
INFO  [main] racewriter: Registered Car [4], waiting for discover of Race

DEBUG [main] simspect: {DISCV} Discovered [ObjectRoot.Car] handle: 4, created
and cached java instance [testcode.racesim.Car] (hash:14554415)
```

Listing 6-12: Remote Object Discovery Log

These log extracts show that objects created in the AOP-model are registered in the federation, and how objects created by remote federates are discovered and cached in the AOP-model, thus demonstrating success for criterion three and four.

<u>Criterion Five and Six: Data Changes</u>

As a pure model runs, changes to the data it has created need to be reflected out into a federation. The log extract below shows Simspect capturing a change to the `distance` field of the `Car` instance that was created by the AOP-model earlier (hash 16453941):

```
TRACE [main] simspect: {F-SET} Field [testcode.racesim.Car.distance] set to
[55.555555555555556] on object (hash:16453941)
```

Listing 6-13: Local Field Modification Log

Realising that this may be interesting information, Simspect then reflects this attribute change out into the federation where it is noted by the companion federate:

```
INFO  [main] racewriter: {REFLECT} Object handle [8] with [1] attributes
INFO  [main] racewriter:    tag: 1199961219096
INFO  [main] racewriter:    attribute: distance, value: 55.555555555555556
```

Listing 6-14: Local Field Modification Being Reflected

Once the data has been successfully introduced into the AOP-model, changes that occur remotely must also trigger updates to the cached data. The log below shows the Simspect federate getting updates to the distance of the Car object it discovered earlier (hash: 14554415).

```
TRACE [main] simspect: {RFLCT} Reflection received. HLA handle: 4, attribute
count: 1
TRACE [main] simspect: {RFLCT} Updated java field [distance] of java object:
(hash:14554415): newValue = 166.66666666666666
```

Listing 6-15: Remote Field Modification Locally Reflected

The log extract above demonstrate success for criterion five and six by showing that information changed locally is reflected out into a federation and that attribute changed on remote instances is received and incorporated into the AOP-model.

Criterion Seven and Eight: Method Calls

As discussed in this chapter, method calls are a common form of behaviour modularisation in object-oriented programming. When providing a solution that merges the HLA and Object Oriented philosophies, interactions are generally used as a rough approximate for method calls. A periphery consideration here is that of data introduction. In a pure-OO model, data is placed into the correct locations (or "introduced" into the model) through the use of situation-specific method calls. To successfully meet criterion seven and eight, the results collected must demonstrate the following:

- Interesting method calls inside the AOP-model are translated to Interactions and sent to the federation where they can be acted on
- Interactions sent by remote federates that represent a method call are received by the AOP-model and cause the appropriate method to be invoked

Data introductions fall under the second category, where a remote federate will use a method interaction to introduce previously created object data to a model. To demonstrate how these criteria have been met, a single scenario will be used, with extracts from the log files of both the AOP-model and the companion federate. This scenario will show an exchange between the two components in the sushi simulation.

Data Introduction Through Interactions

In the first case, log output from the AOP-model shows an interaction that represents a method call being received, and that being translated into a call on the local object:

```
// AOP-model discovers remote object
DEBUG [main] simspect: {DISCV} Discovered [ObjectRoot.Customer.RandomCustomer]
handle: 4, created and cached java instance
[testcode.sushisim.customers.RandomCustomer] (hash:11402211)

// AOP-model processed interaction representing method call
TRACE [main] simspect: {INTER} Interaction received:
type=InteractionRoot.MethodCall.Restaurant_seatVIP, parameters: 2, targetObject:
(hash:4683917)
TRACE [main] simspect:  >>> deserialized handle: 4 to Object: (hash:11402211)

TRACE [main] simspect: {M-INV} public void testcode.sushisim.Restaurant.seatVIP
(testcode.sushisim.Customer) on (hash:4683917)
TRACE [main] simspect:  >>> serialized object (hash:11402211) to HLA handle: 4
```

Listing 6-16: Remote Method Invocation Log

The output above shows the AOP-model discovering an instance of type RandomCustomer and creating a local Java object for it (with the hash 11402211). At this point, the Java object exists, the but AOP-model does not know about it. The rest of the log shows Simspect receiving an interaction that represents the seatVIP method which introduces the previously created object to the pure-model. The argument for that method is deserialized and represents the previously discovered object (with the same hash). The target for this method call is identified as the object with the hash 4683917 (the Restaurant). Simspect then invokes the relevant method on that object, and the invocation is captured by the Aspect (as all invocations are) as shown in the {M-INV} marked section.

This exchange shows how Simspect can successfully convert interactions generated in remote federates into method calls and then invoke those calls locally. It also demonstrates how data is successfully inserted into a model without necessitating an explicit introduction. This successfully meets criterion eight.

To demonstrate how criterion seven is met, the following log file extracts show a method call on a Java object being captured by Simspect and converted into an interaction which is then sent to the federation.

```
TRACE [main] simspect: {M-INV} public void
testcode.sushisim.customers.RandomCustomer.newDishHasArrived
(testcode.sushisim.Dish) on (hash:11402211)
TRACE [main] simspect:  >>> serialized object (hash:7309193) to HLA handle: 43
TRACE [main] simspect: {METHOD}
testcode.sushisim.customers.RandomCustomer.newDishHasArrived: SKIP execution,
object not owned
```

Listing 6-17: Local Method Triggering Interaction Log

This listing shows the Simspect Aspect capturing an invocation of the `newDishHasArrived` method on a local Java object. Realising that this method is of interest, the parameters are serialized (with the `Dish` object replaced by the object handle of the HLA instance that represents it) and an interaction is sent out into the federation. It is also important to note that Simspect identifies that the `Customer` object that the method is being called on is *not* locally managed. It prevents local execution of the method because that could result in local code changing values. This means that any changes that would have been made by the local version of that class will never execute, allowing a remote federate to manage the data changes according to whatever algorithm it wishes to use, not the one present in the local code.

To show that the interaction was successfully sent, the following entries are observed in the log file for the companion federate. They show the interaction being received and the companion federate deciding that the customer is an object it manages and that it should consume the new dish that has arrived:

```
INFO  [main] sushiwriter: {INTERACTION} Class:
InteractionRoot.MethodCall.RandomCustomer_newDishHasArrived with [2] parameters
INFO  [main] sushiwriter: tag: 1199972107843
INFO  [main] sushiwriter: param: targetObject, value: 4 {object_reference}
INFO  [main] sushiwriter: param: dish, value: 43 {object_reference}
INFO  [main] sushiwriter: CHOMP!! Ate dish: name=Mud Cake With Icecream,
type=null, cost=$5.95
```

Listing 6-18: Local Interaction Translated to Method Log

To communicate that the customer has eaten the dish, the companion federate sends a new interaction that represents the method `eat` of the `Dish` class. Back in the AOP-model, this interaction is received and turned into a method invocation. Unlike before, this particular `Dish` instance is managed by the AOP-model, thus, the method invocation is allowed to proceed:

```
TRACE [main] simspect: {INTER} Interaction received:
type=InteractionRoot.MethodCall.Dish_eat, parameters: 2, targetObject: (hash:
7309193)
TRACE [main] simspect:  >>> deserialized handle: 4 to Object: (hash:11402211)
TRACE [main] simspect: {M-INV} public void testcode.sushisim.Dish.eat
(testcode.sushisim.Customer) on (hash:7309193)
TRACE [main] simspect:  >>> serialized object (hash:11402211) to HLA handle: 4
TRACE [main] simspect: {METHOD} testcode.sushisim.dishes.MudCakeWithIcecream.eat
```

Listing 6-19: Interaction Translated to Local Method Log

These results validate that calls within the AOP-model are successfully captured, transformed into interactions and sent out to the federation. They also demonstrate how method invocations on objects that were not created locally are handled (avoiding any HLA ownership problems) and show that criterion seven and eight have been successfully met. However, perhaps most importantly, the series of events presented above provides a clear indication that the approaches embodied in Simspect that allow a pure model and existing federate to act co-operatively together are valid.

Criterion Nine: Results of AOP-model do not match pure-OO model

The final requirement for experiment one, the successful completion of criterion nine requires that all the other criteria also be met successfully. Due to the presence of a remote entity, co-operatively modelling a situation with the AOP-model, the results of executing the model in this scenario should differ from those obtained when executing the pure-OO model by itself. The presence of a remote car in the race car simulation should affect the results of the race. Similarly, the existence of an additional customer in the sushi simulation should cause the distribution of dishes to various customers to change as the remote customer consumes dishes that would otherwise be available when running the model by itself.

To validate the successful completion of this criterion, each of the pure-OO models are run by themselves, with their results noted. It is only after this that the models are run through the Aspect weaver to generate the AOP-model used in the distributed tests.

The Race Simulation

The following output was captured from the two versions of the race simulation. The first extract was taken from the pure model, while the second from the post-weaving, distributed version of the model:

Pure (non-AOP) Model:

```
Starting race...Race Over
[1]: Fastest Car       0:10:00
[2]: Medium Pace Car   0:15:01
[3]: Slowest Car       0:30:00
```

Distributed, AOP-Model:

```
Starting race...Race Over
[1]: RemoteCar         0:05:02
[2]: Fastest Car       0:10:00
[3]: Medium Pace Car   0:15:01
[4]: Slowest Car       0:30:00
```

The differences are clear. When the pure-OO version of the model is run, only three cars (those managed by the model) are entered in the race. When the AOP-model is run in a distributed simulation with the companion federate, the results for the remote car (programmed to be much faster) are included.

The Sushi Simulation

When considering the sushi simulation, the situation is much the same:

Pure (non-AOP) Model:

```
sushi: INFO    Restaurant Simulation Over, No more food left!
sushi: INFO    Closing time: 30.0
sushi: INFO    Table Listing:
sushi: INFO     ->Table 1: customer=CustomerOne(5), availableDish=null
sushi: INFO     ->Table 2: customer=CustomerTwo(6), availableDish=null
sushi: INFO     ->Table 3: customer=CustomerThree(2), availableDish=null
sushi: INFO     ->Table 4: customer=null, availableDish=null
```

Distributed, AOP-Model:

```
sushi: INFO    Restaurant Simulation Over, No more food left!
sushi: INFO    Closing time: 26.0
sushi: INFO    Table Listing:
sushi: INFO     ->Table 4: customer=RemoteCustomer(7), availableDish=null
sushi: INFO     ->Table 1: customer=CustomerOne(0), availableDish=null
sushi: INFO     ->Table 2: customer=CustomerTwo(6), availableDish=null
sushi: INFO     ->Table 3: customer=CustomerThree(0), availableDish=null
```

When the pure-OO model is run, there are only three customers in the restaurant, and each consumes some dishes. However, when the distributed version of the model is run, the presence of the remotely managed customer (who clearly is hungry) alters the results, consuming a large number of dishes that are then not available to the other customers.

From these results it is clear that criterion nine has been met, resulting in the successful completion of this experiment.

6.5 Summary

Throughout the course of this chapter, the design, methodology and behaviour of the Simspect runtime have been presented. Embodied in this discussion has been answers to a number of significant research questions and a description of how pure, object-oriented model code can work co-operatively within a shared, distributed simulation. To conclude, the results for experiment one were presented and discussed.

The solutions presented in this chapter effectively remove the need for HLA knowledge during the model creation process. However, experiment one relied on the hand creation of configuration and mapping data, a process that demands knowledge of the HLA. The next chapter discusses solutions for automating this process, in turn allowing pure models to be automatically transformed into HLA federates.

Chapter 7

Automating Model and Mappings Extraction

Aspect-Oriented Programming provides a methodology that isolates the development of crosscutting, system level concerns, thus allowing them to be implemented separately, keeping core business logic free from such considerations. As discussed in previous chapters, the motivations for implementing HLA-behaviour via such an approach is both beneficial and attractive. Chapter 5 highlighted a number of serious shortcomings that need to be addressed when pursuing this goal. Chapter 6 demonstrated how AOP could be applied to the HLA to enable a true separation of concerns approach to model development.

However, the advances discussed in chapter 6 still necessitate HLA knowledge to manually identify the parts of an OO-model that may be interesting in a HLA environment and to identify where and how HLA considerations should overlap with the model itself. In the proposed environment, this manifests itself in the production of configuration data for a generic HLA Aspect. Such a requirement does not fully meet the goals of this research.

Each pure-OO model contains pockets of information that are of wider interest in the context of a distributed simulation, and accordingly, must be shared with other members of a federation. This is data that is typically codified in the FOM of an HLA simulation. In addition to this "interesting" data, there may be other parts of the model that exist purely to serve implementation specific means, and by itself, has no broader interest. Among other examples, this data might find form as intermediately variables used to cache values during the processing of a specific algorithm.

To remove the manual process that was employed in the previous chapter for defining both a federation object model and the other various configuration elements necessary to allow the generic HLA aspect to function correctly, a method for automatically introspecting a pure model is required.

The notion of automatically introspecting a simulation model in an attempt to extract further information about its underlying data model and the services it can provide is not new. In [129], Yilmaz describes the importance of simulation model introspection in the context of easing the burden of reusing simulation models in situations they were not originally intended for. Yilmaz highlights automatic model introspection as a means for extracting information about the capabilities and requirements of a pure simulation model. This information can then be used to decide if a particular model is fit for use in an entirely different scenario. However, the approach enumerated by Yilmaz also demands that model developers manually add this information to their models during implementation. This requirement represents a mixing of concerns (model development and external requirements/capabilities description) and involves a manual process, thus rendering it mostly unsuitable within the goals of this research.

To address the problem of how model metadata and various runtime artefacts such as configuration information can be created automatically, this chapter employs the basic premiss of model introspection, but goes one step further than Yilmaz, seeking to automate the entire process. Following a discussion on how this can be achieved, this chapter concludes with presentation of the results from experiment 2, validating the proposed automation process.

7.1 Model Introspection

The purpose of the model introspection process is to generate two forms of information necessary for the proper operation of the generic HLA Aspect presented in the previous chapter. Figure 7-1 shows the process used to generate this data.

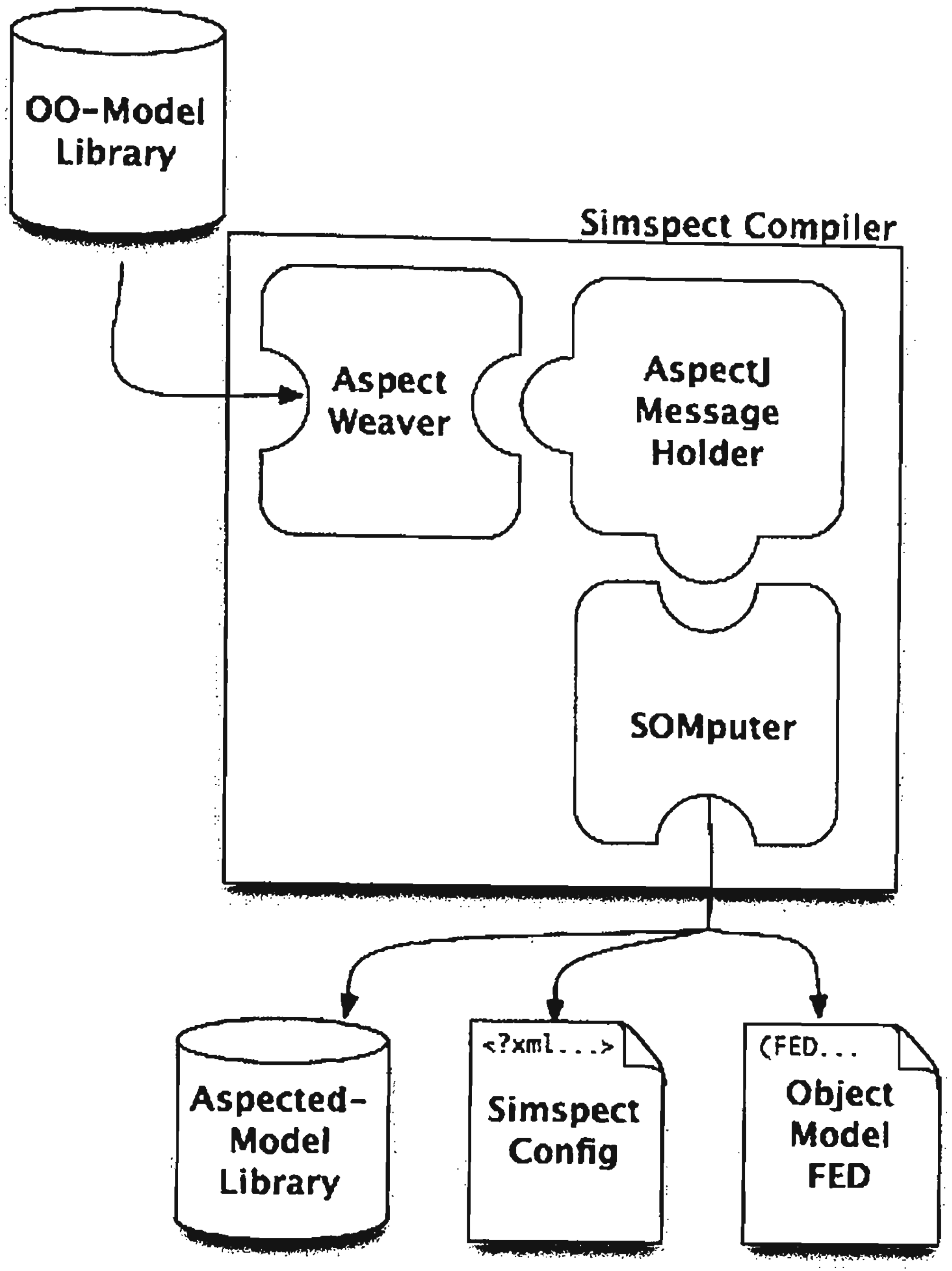


Figure 7-1: Model Introduction Process

To handle the process of converting a pure-OO model into an Aspected-Model (one that has been woven with the generic HLA Aspect from chapter 6), the Simspect Compiler was created. The compiler takes a pure-OO model library (in the reference implementation, this is a Java jar file), weaves the Generic Aspect into the contained code and then outputs an updated version of that library.

Figure 7-1 introduces a new component, named the "Somputer." The object model and configuration data generation is the responsibility of this component. It is instructed to introspect a group of OO classes and to generate the necessary object model and mapping configuration data based on those types.

To make the compilation process more efficient, the compiler will watch over the Aspect weaving process and keep track of which classes it makes modifications to when weaving in the Generic Aspect. Classes that are not touched by the weaver have no way of becoming known to the Simspect runtime, and as such, do not need to be considered when generating mapping and object model data. Rather than having the Somputer process all classes inside the library, only those that have been affected by the weaving process are "somputed" (tested for inclusion in configuration data generation). The *AspectJ Message Holder* shown in the figure 7-1 is the reference implementation component that watches the weaving process and then passes the information it has gathered to the Somputer. When using AOP frameworks that do not provide these types of facilities, the Somputer could be tasked with processing all classes in the library. While not as efficient, this process would still generate functionally complete data.

Multiple HLA Models

Throughout this document I have referred to the need for Simspect to generate an HLA object model from a pure-OO model. As the compiler will generate a different model for each library it processes, it is more correct to think of these as Simulation Object Models (SOM) rather than Federation Object Models (FOM). However, this does raise a potential problem. The output for a single processing of a library is an HLA model document specific to that library. When creating an entire federation from pure-OO models, a number of potentially different HLA model documents will be generated. However, when a federation is created, only a single FOM file can be used, and its contents must be the union of all the SOMs for the participating federates. Unfortunately, this process of taking a group of SOMs and combining them to produce a FOM requires unacceptable levels of HLA knowledge.

This particular problem is more one of implementation, rather than an issue with the underlying theory. One potential solution that could be employed in this scenario would be to extend any Simspect compiler implementation to accept multiple libraries at once (one for each simulation intended for use within a federation). The Somputer would then have the opportunity to generate a HLA model document based on all the pure-OO code used in a federation, not just the subset used by a particular library. In the most recent version of the IEEE 1516 HLA standard (1516-2008), a new feature known as "modular FOMs" has been introduced [71]. This allows each federate to specify its own object model fragment when it joins a federation, at which time the fragment will be used to extend the federation-wide object model. An approach like this could also be leveraged to solve this particular problem.

7.1.1 The Somputer

The role of the Somputer is to introspect the various classes that were modified as part of the AOP weaving process, and to produce the appropriate mappings and object model files based on the parts of those classes that may hold some interest in an HLA context. The specifics of how the Somputer decides what is and is not of interest are covered later in this chapter.

There are four main components that are used within the Somputer to achieved its goals:

- **Object Somputer:** Processes a group of OO classes to assess if there are any interesting attributes that should be mapped onto HLA attributes
- **Interaction Somputer:** Processes a group of OO classes to assess if there are any interesting methods that should be mapped onto HLA interactions
- **Object Model Container Types:** In-memory representation of the HLA object model (and associated mapping data). This is built up by the Somputer during processing
- **Renderers:** Once the Somputer has finished, renderers convert the in-memory Object Model into a persistent form such as configuration file or a HLA fed file

Figure 7-2 shows the internal structure of the Somputer, and the way data flows through its sub-components.

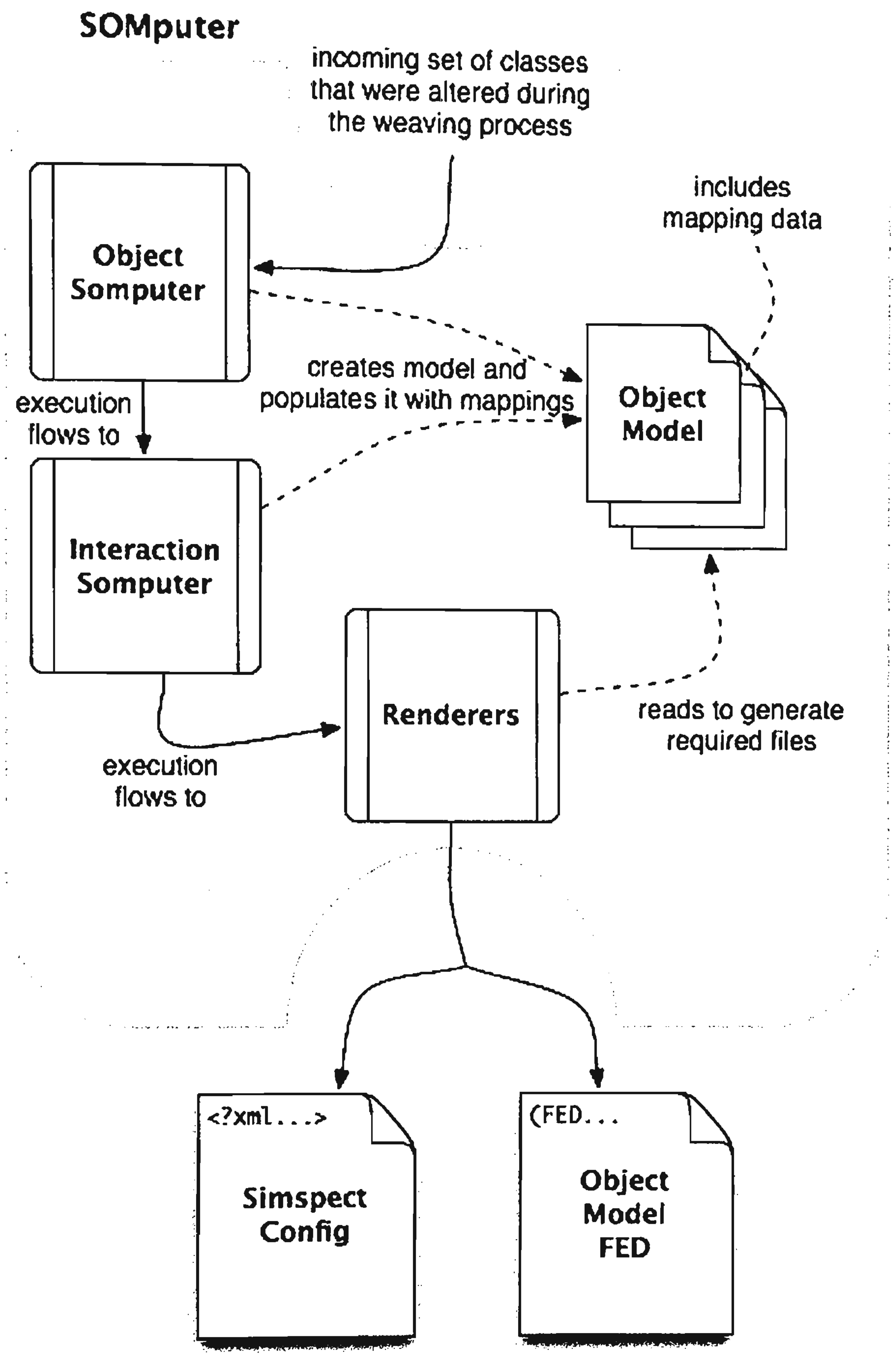


Figure 7-2: The Somputer

Each set of classes given to the Somputer is passed through the *Object Somputer* and then the *Interaction Somputer*. These two sub-components will assess the level of interest in the fields and methods of the classes, populating an in-memory Object Model hierarchy. Once this is complete, a group of *renderers* is given the task of converting that hierarchy into the relevant persistent forms. In the case of the reference implementation used during experimentation, this includes the production of an XML configuration file (containing mapping data) and a HLA object model fed file that can be used to create a federation.

If the reference framework were to be implemented with a different set of technologies, additional renderers could be created to fulfil any further requirements (perhaps the generation of middleware code for an environment with less reflective capabilities than Java).

7.1.2 Storing Mapping Data

The Object Model container types created as part of the somputation process and used by the renderers are designed to capture two types of information:

1. Information necessary to build a complete HLA object model
2. Information about how the parts of that object model are related to the pure-OO model

As processing unfolds, the Somputer attempts to build a HLA object model that mirrors as much as possible the hierarchy present in the pure-OO model. Figure 7-3 shows how the container types link up generated HLA object model data with the pure model types.

Figure 7-3 contains two sets of class hierarchies. On the left, the structure of the pure-OO model shows a total of four classes, each with varying numbers of attributes. On the right is the HLA model being built by the Somputer. Mappings data types work by linking together the relevant HLA type from the HLA object-model, and the Java type that represents it in the pure model. Once the Somputer has decided that a particular class or attribute should be included in the HLA model, it will create a mapping to maintain this association. When finished, this information is written to a configuration file by a renderer. From here, the Simspect Runtime can reconstitute the data during a distributed simulation.

It is also important to note that not all classes or attributes that exist in the pure model have a representation in the HLA model. The Somputer will only add an HLA class or attribute to the model when it has decided that the entity will be of interest in a HLA setting. In Figure 7-3, only some of the attributes from the OO-model are present in the HLA model, and one class is missing entirely.

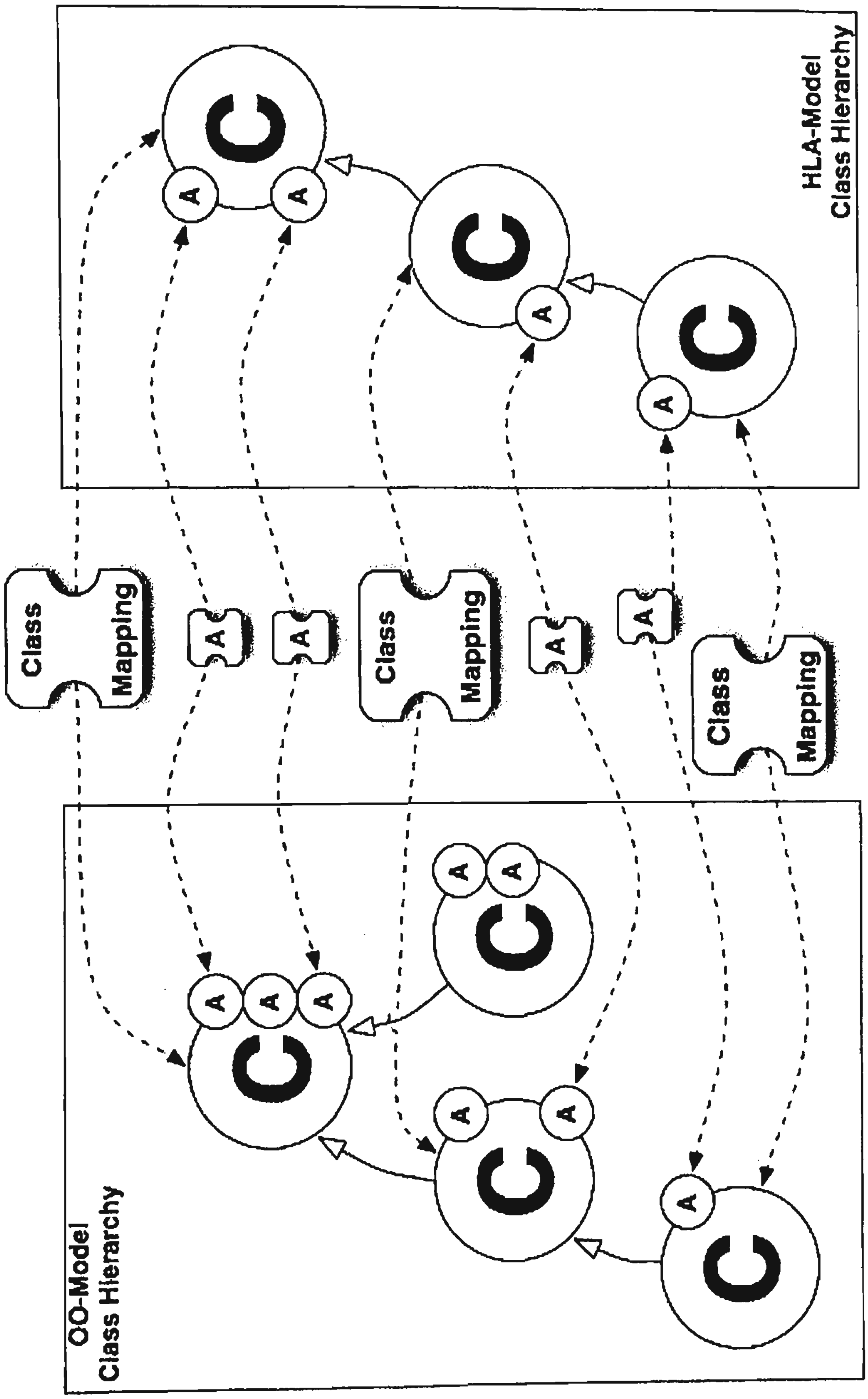


Figure 7-3: Object Mappings

The reference implementation uses Java reflection types in the mapping data to identify parts of the pure-OO model that relate to the HLA types. In environments where reflective capabilities are not directly available, it would be the responsibility of the developer to create some kind of analogue that could be used to uniquely represent parts of the OO-model hierarchy.

The general theory for method mappings is similar to that of object mappings. However, unlike object mappings, the inheritance hierarchy implemented in the HLA model is not designed to mirror the hierarchy in the pure-OO model. Much like object classes, HLA interaction classes have an inheritance hierarchy, with parameters of a particular class being inherited by all its children. This is quite different from the structure of methods that exist in OO. Also unlike OO, HLA interactions have no association with a particular object class. Interactions are designed to function more like transient messages.

To provide the kind of semantic connection necessary to have interactions represent method calls, a special interaction class is defined. This class has a single parameter that identifies the target object of the method call represented by the interaction (discussed previously in chapter 6). When generating an HLA object model and method mappings, the Somputer will create individual interaction classes that extend from the predefined type, and which provide additional parameters according to the parameters defined in the OO model.

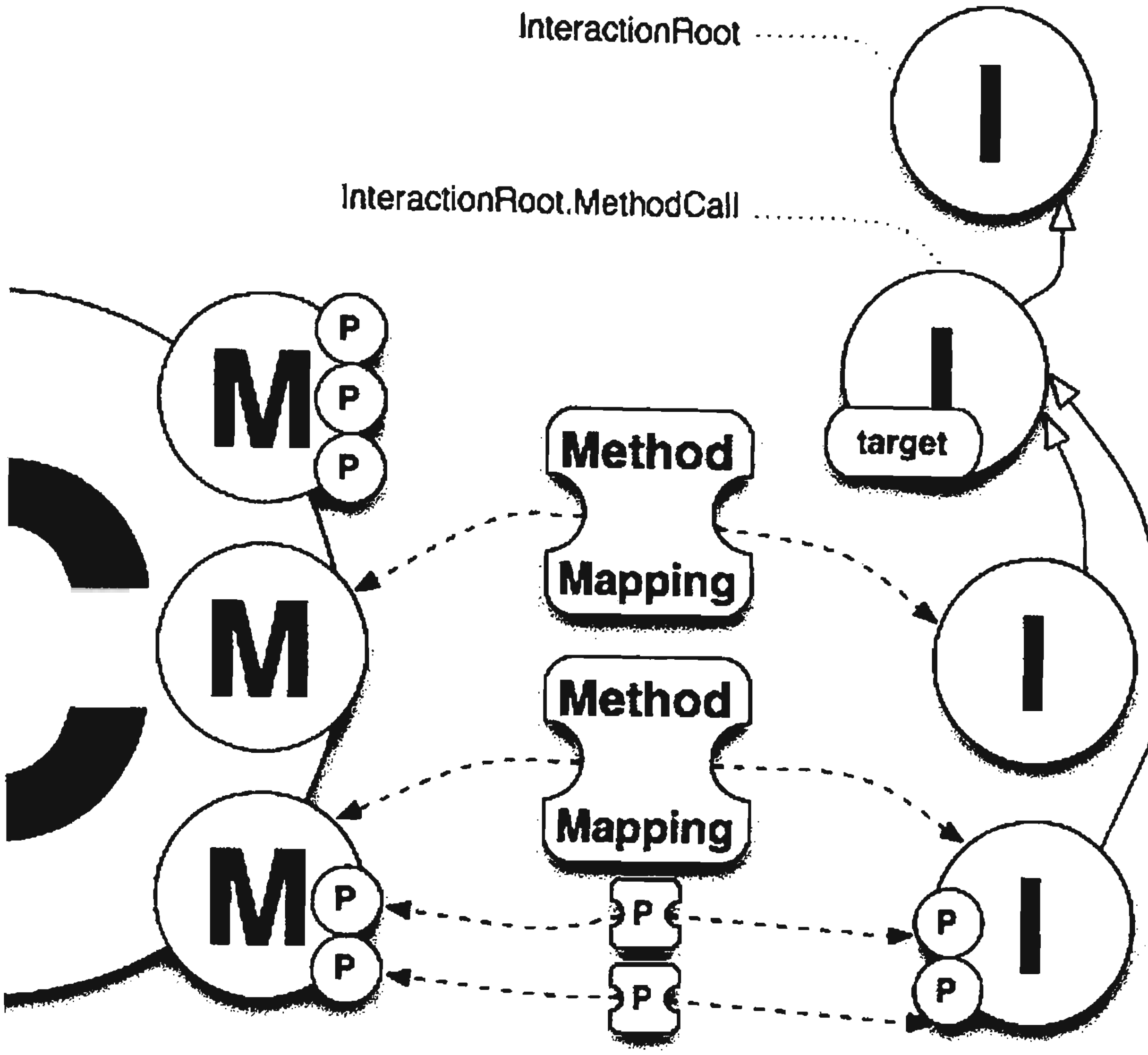


Figure 7-4: Method Mappings

In Figure 7-4 there are two methods that the Somputer has deemed of interest (the process used to make this decision is discussed later in this chapter). For each of these types, an interaction class with the appropriate additional parameters has been created and method and parameter mappings have been generated to provide the necessary linking information. For more information on how method calls work, see section 6.3.3.

7.1.3 Storing Type Information

To allow the Simspect runtime to properly serialise and deserialize information between its local representation (in this case, Java) and its HLA representation, type information must be recorded in configuration data. Type conversion information is maintained in each attribute and parameter mapping. In the reference implementation, the type information is represented as a simple enumeration. The table below outlines each of the values and how they correspond to the Java type in the reference implementation.

Enumeration Value	Java Type	HLA Type
BOOLEAN	boolean	boolean
CHAR	char	char
BYTE	byte	byte
SHORT	short	short
INTEGER	int	int
LONG	long	long
FLOAT	float	float
DOUBLE	double	double
STRING	java.lang.String	char[]
REFERENCE	Any non-String object	int (object handle)
REFERENCE_ARRAY	Arrays, Lists, Sets	int[] (object handles)

Table 7-1: Simspect Enumeration Mappings

When performing conversion at runtime, Simspect uses a set of encoding helpers provided with the RTI implementation. In this way, any type (such as an integer or String) is converted into the opaque byte array format necessary for use with the HLA. Object references are passed via the HLA as the handles of the objects that are being referenced. Where the Java type is any sort of collection (array, list, etc...), an array of integers representing the handles of the referenced objects is used. As discussed in the previous chapter, Simspect forgoes any of the advantages that the IEEE 1516 specification provides with regard to specifying the structure of complex data types. While these additions are valuable, they do partially defeat the purpose of the HLA as individual parts of those types cannot be independently published or subscribed to.

Having shown how mapping information is structured, it is now time to discuss the decision processes used by the Somputer when constructing an HLA object model (and associated mappings) from a pure-OO model.

7.2 Introspecting Objects

When a class is passed into the Somputer, each directly declared (non-inherited) field is inspected to see if it contains any attributes of interest.

It is important to note that the somputation process is not quite as simple as passing a series of classes through the object and interaction somputers and then rendering the generated data at the end. The Somputer has a tendency to jump around from class to class based on the data it finds and the requirements those findings create.

For example, to keep the object hierarchy consistent, if a particular class is adjudged to contain an interesting attribute, all the parent classes of that class are forcefully somputed and added to the generated object model. In some cases, the Somputer will have previously processed these classes and decided that they have no interesting attributes (and thus skipped them). However, the requirement to maintain the object hierarchy overrides this, forcing the class to be processed and added to the model. This particular process is especially important where abstract parent classes are used to provide a common base to children. Direct instances of these classes can never actually exist, leading one to reason that they do not need to be present in the HLA model. However, including them means that other federates can subscribe to the type and thus hear about an entire subset of classes without having to subscribe to each individual one.

Consider the Sushi simulation: external federates might be interested in hearing about the presence of any dessert dishes that enter the simulation. Although the `Dessert` class contains no interesting attributes of its own, including it allows federates to get hear about all the concrete dessert dishes without needing to know each and every concrete type that exists.

It is these types of requirements that mean the Somputer jumps from class to class when processing and classes that have previously been defined as not interesting may actually end up in the generated model. This also means that the algorithm used to process a class is quite complex.

7.2.1 What Makes an Attribute Interesting?

As stated throughout this chapter, the only attributes included in a generated model are those deemed "interesting" from the perspective of the HLA. To provide some background to the problem, this section will discuss the basic concepts of what defines an attribute as interesting in an HLA context before delving into the specifics of the algorithm used to automatically determine this status (the Object Somputation algorithm).

When creating a pure model, the process of designing and building an object-oriented class has to take into consideration many factors. Although the realisation of a particular class will contain attributes that are specific to the entity being modelled, in many situations there will also be a number of other data values included purely to support the implementation model chosen, to make some processing simpler or more efficient, or to make the application easier to maintain. Put another way, each OO class may contain a number of attributes that do not relate to the entity being modelled.

In the previous chapter, a manual process was used to subjectively identify these types of attributes and separate them from those that form part of the core entity being modelled. Core attributes had mappings configuration data and object model entries created for them by hand, whereas non-core attributes were ignored. The purpose of the Somputer is to automate this identification process. Thus, some definition of how a core or "interesting" attribute is recognised is needed. To this end, the Somputer assesses each attribute for worthiness according to the following process:

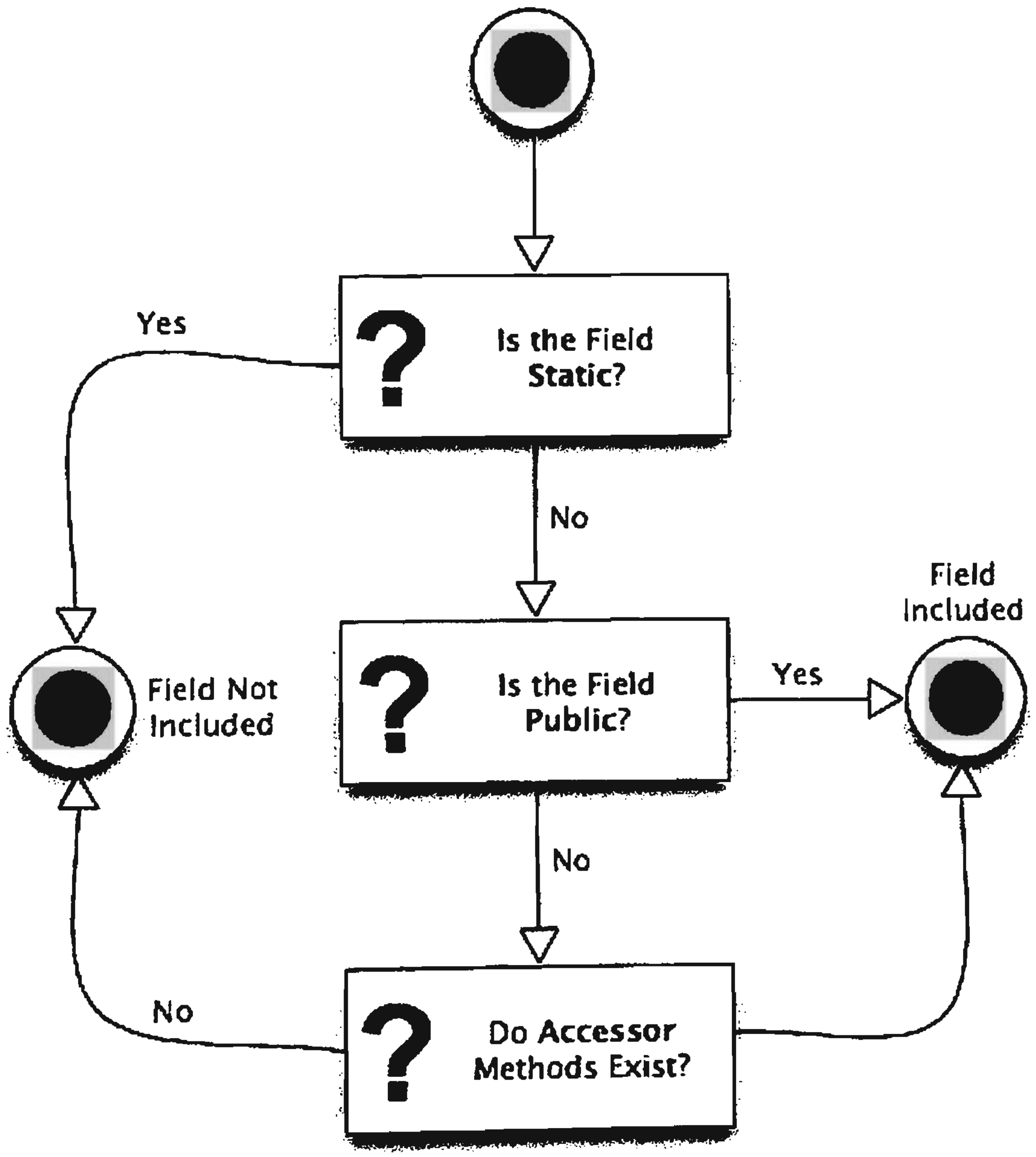


Figure 7-5: Field Assessment

As shown in figure 7-5, if the field is static (a class-level attribute) it is immediately ruled out as uninteresting. In the HLA, all data is directly attached to a single instance of an object class, no notion of a static variable exists. Although it is entirely possible that this may be model data worthy of inclusion within the FOM, there is no clear mechanism through which it can be represented in the HLA. For this reason it is ruled out as useable data.

Following this, the access level declared for an attribute is taken into consideration. If the field is marked as public, it is immediately accepted. As other model elements have access to the `public` fields of another class, such an attribute could be validly assumed to have some interest throughout the model. If this were not the case, and the attribute were for internal processing purposes, it would most likely have been marked as `private`.

It is widely recognised as good OO practice for instance variables to be assigned `private` access, with accessor and mutator (get and set) methods provided as the means through which to gain and alter their values. Accordingly, if an attribute is marked as `private` or `protected`, further processing must be done to assess if the field is interesting or not. The approach taken by Simspect is to search through all the declared `public` methods of a class looking for accessors.

The reference implementation makes use of the Java programming language, and in Java, the common idiom is for accessors to take on the form `getXxx()` and mutators to take the form `setXxx()`, eg. The field `Car` would correspond to `getCar()` and `setCar(Car)`. If a field does not have a corresponding accessor method, it is deemed uninteresting and skipped. Without an accessor there is no way other model elements could extract its value, and thus, one can infer that it is not meant to be part of the public object model.

Edge Cases, Algorithm Improvement and Scope

The presence of a mutator method is not considered when accessing an attribute. Although such a method would allow other model elements to change the value of the attribute, this type of cross entity interaction is foreign to the HLA. The strict ownership rules present in the HLA specification mean that it would be rare for one federate to directly alter the value of an attribute of an object instance created by a different federate. Although the specification allows for the transfer of attribute ownership (and with it, the privilege to update the value of an attribute created elsewhere), these facilities are colloquially accepted to be among the least frequently used parts of the HLA specification. For reasons of scope, ownership considerations have been omitted from this work.

However, the crossover between the HLAs ownership rules and Object Orientation would be an interesting area for further research.

Also not considered by this algorithm are synthesised variables. That is, values that are externally presented as a single field (through the presence of an accessor and mutator) but are implemented internally using more than one variable (or perhaps even none at all). It would be a reasonable approach to make the assumption based on the public interface of a class that the presence of a public accessor signals some useful piece of data is being provided (and should form part of the corresponding HLA model). However, determining when that value changes, and accordingly, when the HLA federation should be notified of such a change, presents significant problems.

Some way to identify which attributes provided the backing data for synthesised variables would first be required. Along with this, some understanding of how to capture changes to these variables is also needed. If the variables reside in different object instances, the relationship between these objects must also be considered. Further, some consideration of how best to handle public attributes whose synthesis depends purely or partly on non-attribute backing (perhaps the result of some algorithm) is also needed, along with some understanding of how to programatically assess when changes occur in such a situation. Given these problems and that synthesised variables are a relative edge case, their consideration is deemed beyond the scope of this work.

As with any automated process, the approaches in this chapter could result in the misidentification of interesting or non-interesting fields. Without explicit knowledge of the model designer's intent, it is difficult to make correct decisions in all situations. The primary goal of this work is not to develop a foolproof algorithm for automatically identifying fields that should form part of a HLA FOM, but rather to demonstrate that an automated approach is possible and to show how the process of linking OO and HLA models can be achieved after those decisions have been made. Any algorithm that is developed could be continually refined over time. Experimentation discussed later in this chapter has shown the approach above to be effective in test cases that are designed specifically to cover a number of common and potentially tricky OO scenarios. Deeper investigation and expansion of these approaches is a fertile place for further research to begin, and any enhanced algorithm could be easily integrated into the broader framework introduced by Simspect.

7.2.2 The Object Somputation Algorithm Explained

Having discussed how an individual field is assessed to decide if it should be included in an HLA object model, some attention must be given to the process involved in deciding whether an entire class is interesting or not. Classes deemed as interesting need to be included in generated HLA object models, while those that are not are left out.

There are several factors that define whether or not a class is interesting. The primary measure of interest is the presence of attributes that are themselves considered interesting. However, a class that contains no interesting attributes of its own might still be required in the generated object model. To keep the structure of the HLA model as close to that of the OO model from which it was derived, classes that would individually be deemed uninteresting may still need to be present. Maintaining the inheritance hierarchy present of the OO model is one possible reason for this. As general OO theory would support, HLA federates subscribing to an object class within a FOM may receive object discovery notifications for instances that were registered at a more specific type. Although such federates cannot access the full state of the object (as they as subscribed only to a parent class) they still receive notification. For example, a federate wanting to know about the existence of *all* Vehicles in a simulation might wish to subscribe to a `Vehicle` class. If the Vehicle class has no interesting attributes it would normally be left out of a model. However, it may have subclasses that are interesting. Given this, it is perfectly reasonable to expect that a given federate may wish to subscribe to the Vehicle class (despite it being uninteresting when assessed individually) and therefore, it is important to carry the same hierarchies over from an OO model into a generated HLA model.

The entire process required to assess a particular class is quite extensive. The primary decision tree is provided in Figure 7-6, however, this figure references subsequent diagrams where the process branches off.

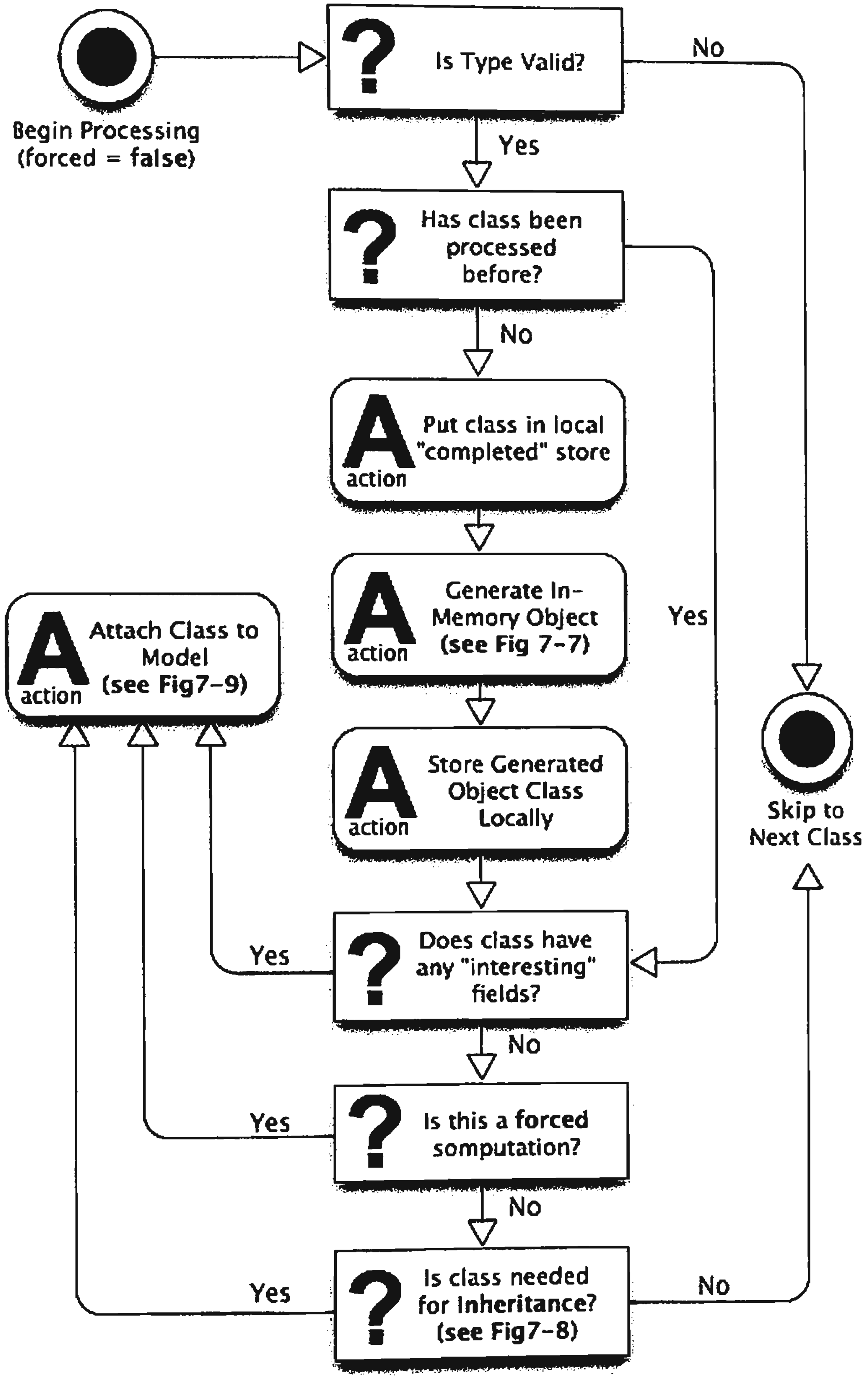

Begin Processing
(forced = false)
Is Type Valid?
No
Yes
Has class been processed before?
No
Yes
action
Put class in local "completed" store
action
Generate In-Memory Object (see Fig 7-7)
action
Attach Class to Model (see Fig7-9)
action
Store Generated Object Class Locally
Skip to Next Class
Yes
Does class have any "interesting" fields?
No
Yes
Is this a forced somputation?
No
Yes
Is class needed for Inheritance? (see Fig7-8)
No

Figure 7-6: Object Somputation

Valid Types

The first step is to assess if the class represents a valid type. This check is designed to stop undesired or "unmappable" types from entering the generated model. The exact list of types that fall into this category will be dependent on the implementation platform in use. In the reference implementation (which makes use of Java) there are a number of types that are excluded from being valid.

Interfaces are excluded because they do not represent a concrete type, and therefore instances of it cannot be directly created or enter into the simulation. Java Enumerations are also excluded as they do not form part of the object class hierarchy. It makes little sense to subscribe to a particular enumeration. The more common use would be for *attributes* to take the value of an enumeration, rather than having federates subscribe to an enumeration type[21]. This is the same reason primitive types (or their Java class analogues) are also excluded.

Finally, certain classes are also selectively ignored as dictated by the platform. For example, in the reference implementation, any classes that make up part of the Java Development Kit itself are ignored. The goal of Somputation is to introspect and generate a model from user developed code, not that of the JDK. Further, any class with a main method is also excluded as this signals that the class most likely only exists for the purpose of starting the application, not for use as part of the model. The particular types that fall into this category could easily be contracted or expanded according to the requirements of the situation or platform. The salient detail is to recognise that it may be necessary to perform this type of examination.

In-Memory Objects

Assuming the type is valid, a check is made to see if the class has previously been processed. This is necessary to avoid problems with infinite looping in situations where there is a circular dependency between two types. If the class is marked as having already been processed, it is skipped. Once it has been determined that this is the first time a class is being assessed, information about the class is placed in the "completed" collection. This collection was used in the previous check and placing an entry in here now (even though the class has not been completed) avoids the infinite loop problem. If the class has been

[21] The IEEE 1516 standard provides specific support for enumeration's as data types. There is an obvious overlap that could exist here. As mentioned earlier, this thesis only concerns itself with the HLA 1.3 standard. Further work could explore how to realise such an environment with the IEEE 1516 standard.

processed, execution skips to the part of the algorithm where suitability for the final object model is assessed.

The final product of initial processing is an in-memory representation of the generated object model that can then be rendered to a file. All valid classes that are parsed produce in-memory representations, regardless of whether they are considered interesting or not. However, only those classes deemed necessary are patched into the final object model (this is discussed shortly).

To create the in-memory representation, the class is introspected, searching for any interesting features. In the reference implementation Java reflection is used, in an environment with less introspective capabilities, the raw source code for a class could be parsed to obtain the necessary data. In-memory representations are generated for each valid class. The information extracted is then used to determine if the class is interesting or not.

As shown in Figure 7-7, the first step is to create an `ObjectClass` object for the class. This is the container that stores Simspect reflective information. It has a collection of interesting attributes and a link (initially empty) to its parent class and any child classes. The somputation algorithm will iterate through each attribute declared by the class and will judge whether or not each is interesting using the approach presented earlier in this chapter. Inherited attributes are ignored. They are process when assessing the class in which they are declared.

For each interesting attribute an `AttributeClass` is created to store field metadata. Included in this data is the type of the field. For each type there is a mapping that defines the relationship between the OO type and the HLA representation (see Table 7-1). If the type of a field is a `REFERENCE`, the class referred to must also be assessed so as that logical connection can be maintained in the HLA representation, ensuring that the type is present in the generated FOM. As such classes are needed by this interesting attribute, that class *must* appear in the final object model, so the forced flag (discussed below) is set to true and the type is assessed. The same is true for `REFERENCE_ARRAY` types, however in this case it is the base type of the collection - the type of objects stored in that collection - that is assessed. In this situation, the reference implementation has a particular shortcoming. Due to the constraints of Java reflection, the base type of a collection can only be determined for arrays. For collection types such as Sets and Lists, it is not possible to determine this information. In this particular scenario, a source-level parser could potentially be used to overcome this limitation.

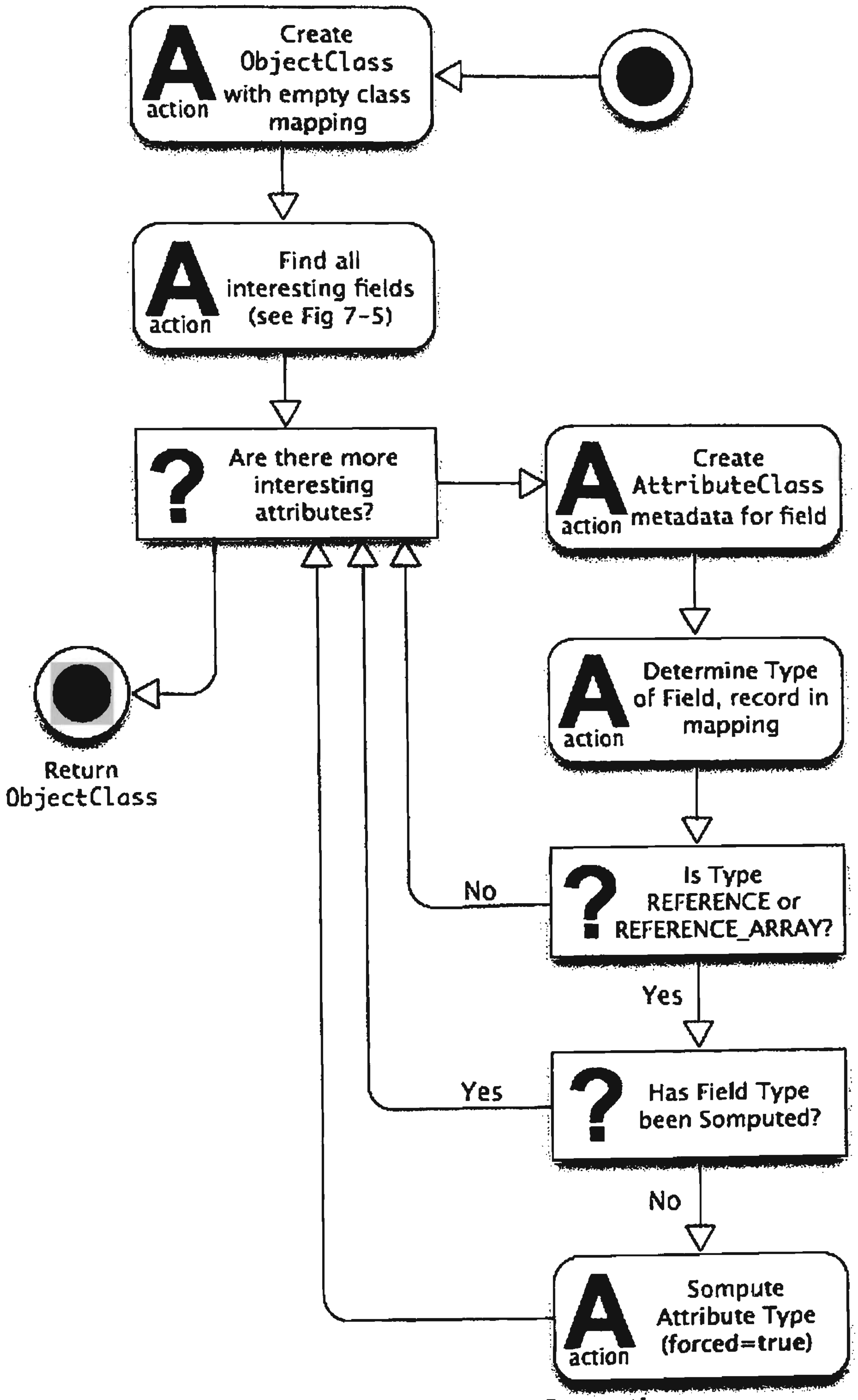

Create ObjectClass with empty class mapping
action
Find all interesting fields (see Fig 7-5)
action
Are there more interesting attributes?
Create AttributeClass metadata for field
action
Determine Type of Field, record in mapping
action
Return ObjectClass
No
Is Type REFERENCE or REFERENCE_ARRAY?
Yes
Yes
Has Field Type been Somputed?
No
Sompute Attribute Type (forced=true)
action

Figure 7-7: ObjectClass **Generation**

Determining Interest in a Class

Having generated an in-memory representation, the Somputer now has access to all the Simspect-relevant meta-information about a particular type. This data can now be used to decide if the type should be included in the final HLA model or not. There are many ways a class can be adjudged as relevant, and the first step is to look at its contained fields. During the previous step, `AttributeClass` instances will only have been created for field deemed interesting (according to the method presented in section 7.2.1). If a class contains at least one interesting attribute, it is attached to the generated object model (this process is discussed below).

Should a class not contain any interesting attributes, two further checks must be made. As highlighted earlier, even if a class is not individually interesting, it might still need to be present in the HLA object model. When processing of a class begins, a flag is provided that signals whether the class should be forcefully attached to the final object model or not. This is used in situations where the algorithm deems a class as necessary despite the lack of interesting attributes, such as when it is specified as the type of an attribute in another class. If the provided flag is switched on, the object is attached to the model without question. Assuming that this is not the case, the final check establishes whether the class is needed to maintain the appropriate inheritance hierarchy.

Abstract Inheritance and Uninteresting Classes

The final check involves traversing the inheritance hierarchy of the particular class. The process only occurs when the class itself either has no interesting attributes of its own or is not subject to a forced somputation. In this situation, all the parent classes of the given class are inspected to see if any are both abstract *and* contain interesting attributes.

If a class has no interesting attributes of its own, it generally has no place in the HLA model. Indeed, if this check is even run it indicates that up until this point the Somputer has not been able to find a compelling reason to include it. However, if a non-interesting class has a parent that is abstract, and that class contains interesting attributes, it is necessary to include the entire hierarchy in the final model. The reason for this is that an abstract parent with interesting attributes indicates that the interesting part of the child class is the specialisation it provides. Consider the Sushi simulation used in experimentation. Interesting information is held in the `Dish` class, which is abstract. Although the child classes that represent the actual dishes have no interesting attributes of their own, the fact that each dish is represented by individual classes indicates that the type of class is itself interesting information. Indeed, it is the way I tell that two given dishes are in fact different types of food. The class specialisation itself is an interesting piece of information making each type worthy of inclusion within the FOM. Any time an abstract class with interesting attributes exists, this type of situation can arise.

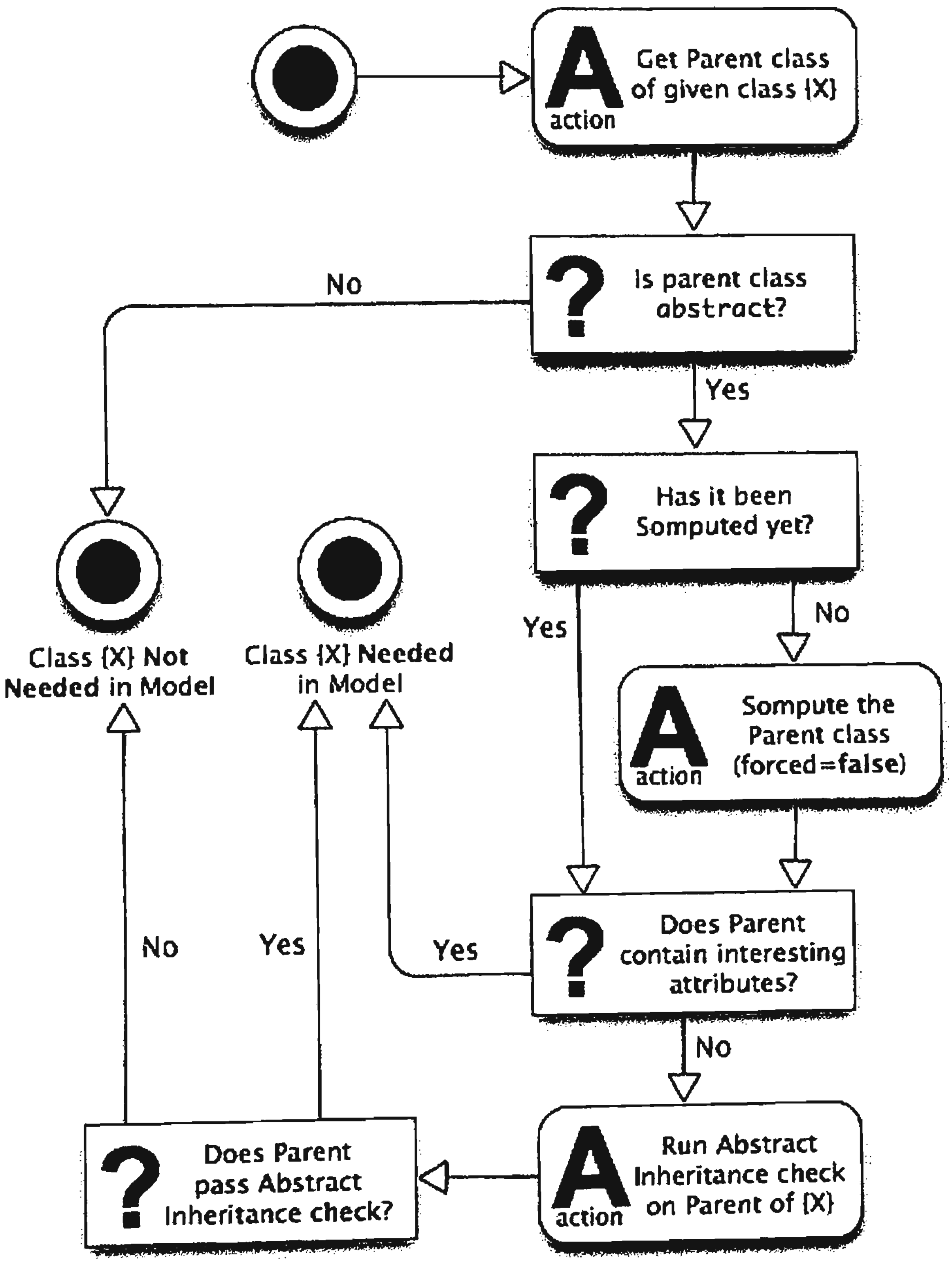
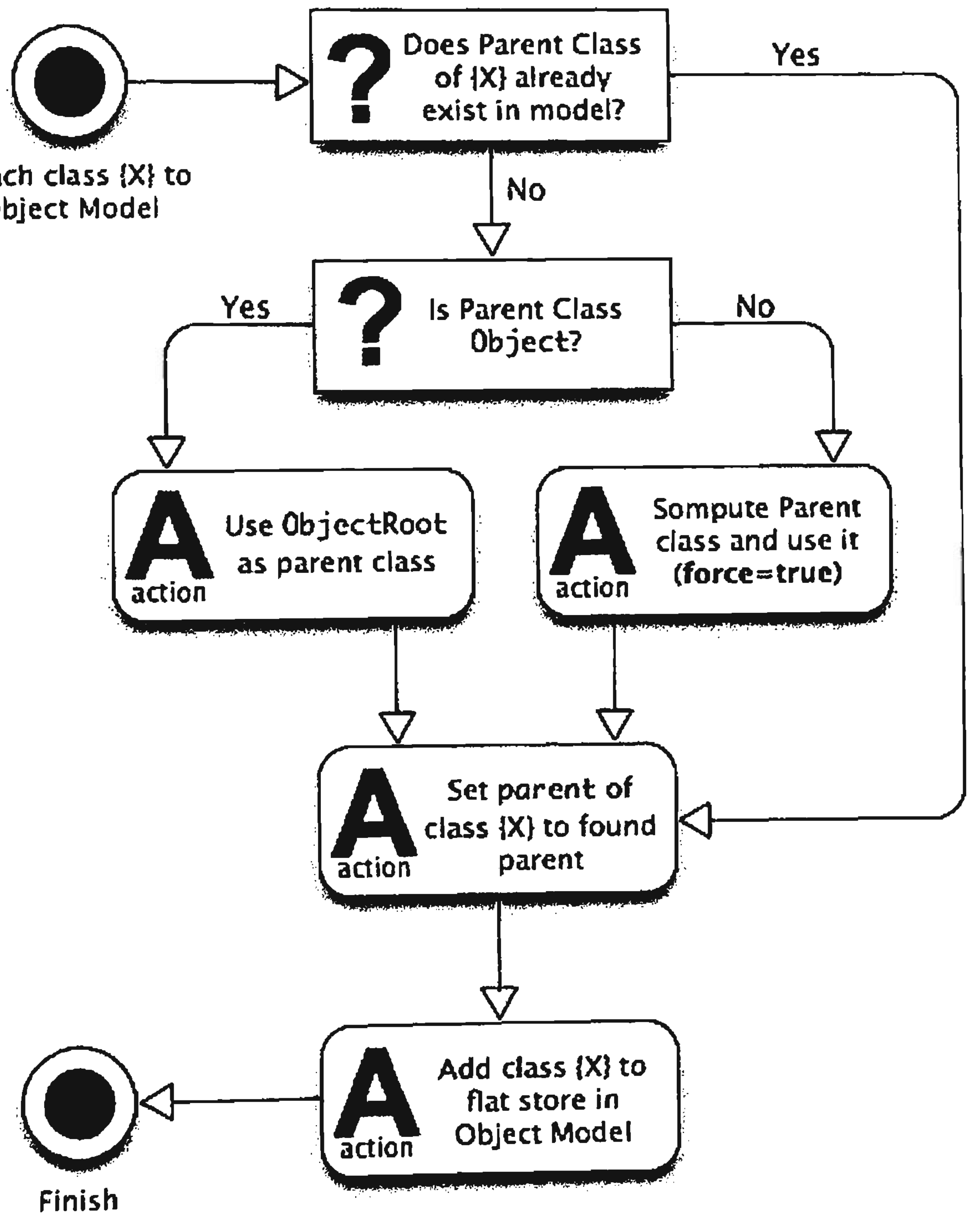

A action Get Parent class of given class {X}
? Is parent class abstract?
No
Yes
? Has it been Somputed yet?
Yes
No
A action Sompute the Parent class (forced=false)
? Does Parent contain interesting attributes?
Yes
No
A action Run Abstract Inheritance check on Parent of {X}
? Does Parent pass Abstract Inheritance check?
No
Yes
Class {X} Not Needed in Model
Class {X} Needed in Model

Figure 7-8: Abstract Inheritance Assessment

The abstract inheritance check searches for this particular situation. If it is found, the child type is deemed to be interesting based on its specialisation, and is forced into the final object model.

Attaching a Class to the HLA Model

Once it has been determined that a particular class is interesting (by any of the means previously discussed), that class must be marked as part of the generated model. Figure 7-9 shows how this is achieved.



Attach class {X} to Object Model
Does Parent Class of {X} already exist in model?
Yes
No
Is Parent Class Object?
Yes
No
Use ObjectRoot as parent class
action
Sompute Parent class and use it (force=true)
action
Set parent of class {X} to found parent
action
Add class {X} to flat store in Object Model
action
Finish

Figure 7-9: Attach Class to Object Model

As mentioned earlier, in-memory representations are generated for all classes that are Somputed. Each `ObjectClass` has within it a slot to identify its parent. This is used to note where in the HLA object model hierarchy a particular class fits. At the conclusion of somputation, any classes that do not have a parent are excluded from the final model. If a class is deemed interesting, it would have been patched into the model and its parent would have been identified.

At the beginning of somputation, an instance that represents the `ObjectRoot` of the HLA model is manually created. This type is considered analogous to the parent of all classes in the OO model (in the case of Java, a class explicit exists in the form of the `Object` type). This manually created class is used as the parent for any classes that directly inherit from the parent of all types.

If the parent OO class is not the object root equivalent, the next step is to see if the Somputer has already processed the parent. If it has been, the resulting in-memory type is set as the parent of the incoming type and processing finishes. If the type has not yet been processed, Somputation of the parent OO class will take place. Further, this Somputation will be **forced**. To maintain the proper inheritance hierarchies across the OO and HLA object models, *all* parent types of any class that is deemed interesting will be included. This forced Somputation achieves this.

Note that this is distinct from the actions that are involved in the abstract inheritance check. That process simply checks to see whether or not a type should be included in the final model (which may trigger this attachment operation). This particular activity involves walking the inheritance tree for a type that has already been deemed interesting. Also note that each class is only fully-processed once. The forced Somputation of a parent class that has already been processed will only trigger its attachment to the model. As the in-memory representation has already been generated, that particular step it skipped.

7.2.3 Completed Object Model

Following the complete Somputation process, a full hierarchy of interesting classes will have been built up underneath the manually created `ObjectRoot`. However, during this process, another form of calculation is also occurring. Once each class has been inspected for its level of interest with regard to the attribute data it contains, its methods are also scrutinised to determine if they are suitable for mapping to HLA interactions.

7.3 Introspection Methods

The way in which the Simspect framework represents methods in the HLA was presented in section 6.3.3. Having finished assessing a class to extract attribute information, attention turns to each of the methods it declares. The process used to assess whether or not a method is suitable for mapping to an HLA interaction is introduced in Figure 7-10:

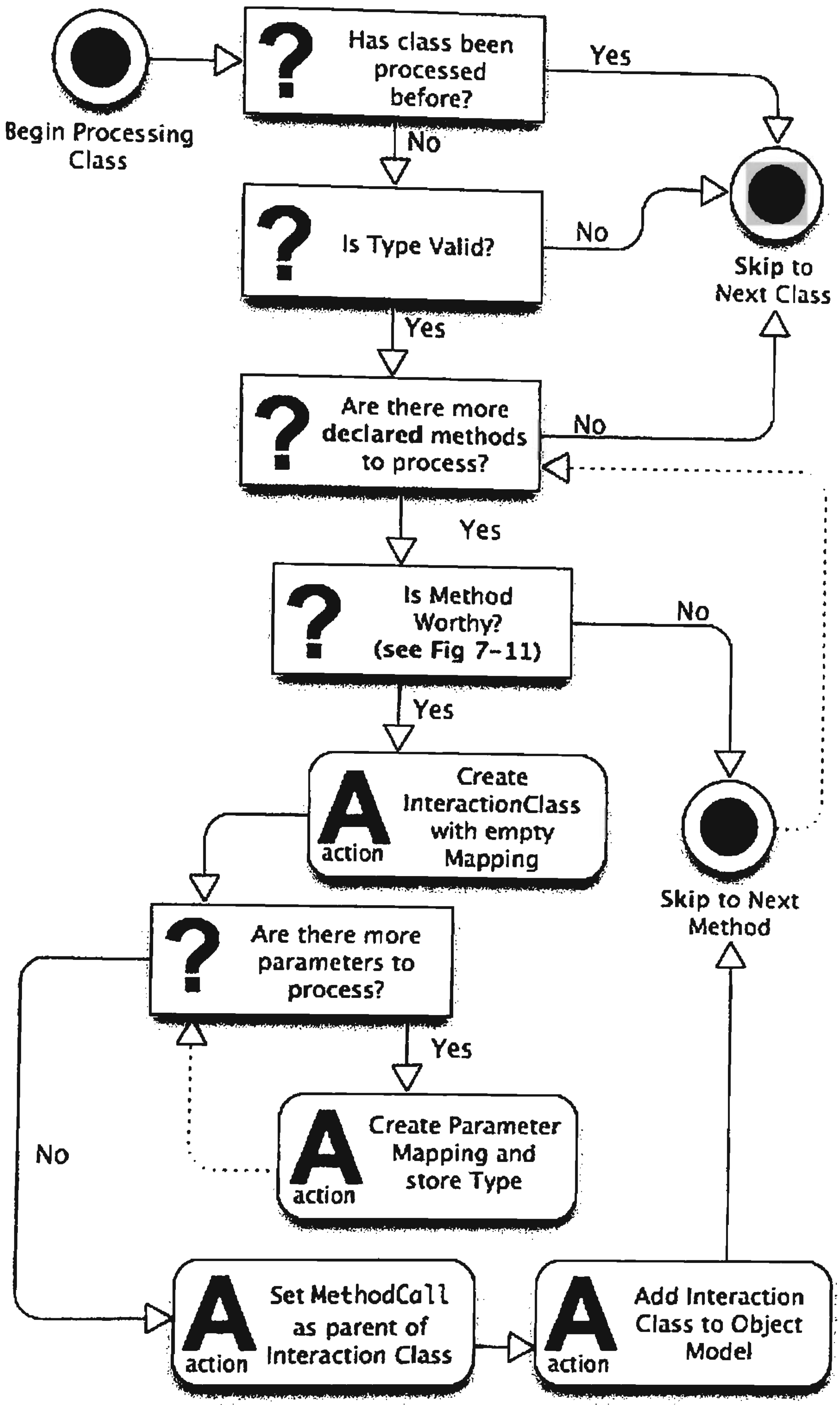

Begin Processing Class
Has class been processed before?
Yes
No
Skip to Next Class
Is Type Valid?
No
Yes
Are there more declared methods to process?
No
Yes
Is Method Worthy? (see Fig 7-11)
No
Yes
Create InteractionClass with empty Mapping
action
Skip to Next Method
Are there more parameters to process?
Yes
No
Create Parameter Mapping and store Type
action
Set MethodCall as parent of Interaction Class
action
Add Interaction Class to Object Model
action

Figure 7-10: Interaction Somputation

The first two steps are the same as for assessing a class for attribute data. If the class has been processed for interactions before, it is skipped. If the class does not represent a valid type, as judged by the same criteria used previously, it is also skipped.

Each method declared but not inherited by the class is inspected to see if it is worthy of inclusion within the FOM. Should it be judged that the method meets all the appropriate criteria (discussed below), an empty `InteractionClass` and associated `MethodMapping` will be created. The mapping stores a reference to the pure model method and to the interaction class metadata that will represent the HLA interaction. Each of the parameters to the method is then evaluated.

Creating Parameter Mappings

For each parameter of an interesting method, a new mapping is needed. This mapping records the parameter name[22] and the mapping type of the parameter (see table 7-1 in section 7.1.3 for valid types). There will inevitably be some parameters whose type is another pure-model entity class. Accordingly, if instances of these entities are going to be referenced by parameters, the model types need to be mapped to the HLA and represented in the FOM. If the type of a parameter has not been Somputed for attributes, or has not yet been included in the FOM, its presence as the type of a parameter will cause this to occur. The parameter assessment process will invoke the object somputation algorithms for the particular type and will set the *force* flag to true, thus ensuring the type is mapped into the generated FOM.

The final step in the method assessment process is to locate the parent interaction class to attach the newly created `InteractionClass` metadata object under. As proposed in section 6.3.3, all method calls exist in a flat hierarchy under a single parent: the `InteractionRoot.MethodCall` interaction. Once this is completed, computation moves on to assess whether or not the next method is interesting.

Identifying Interesting Methods

Section 6.3.3 introduced the characteristics of methods that can be mapped to HLA interactions. Figure 7-11 shows the process used to determine whether or not a method falls into the "interesting" category and necessary for inclusion within the generated FOM.

22 In the reference implementation, the name is actually auto-generated based on the position of the argument. This is due to a limitation in Java reflection whereby attribute names are not available at runtime.

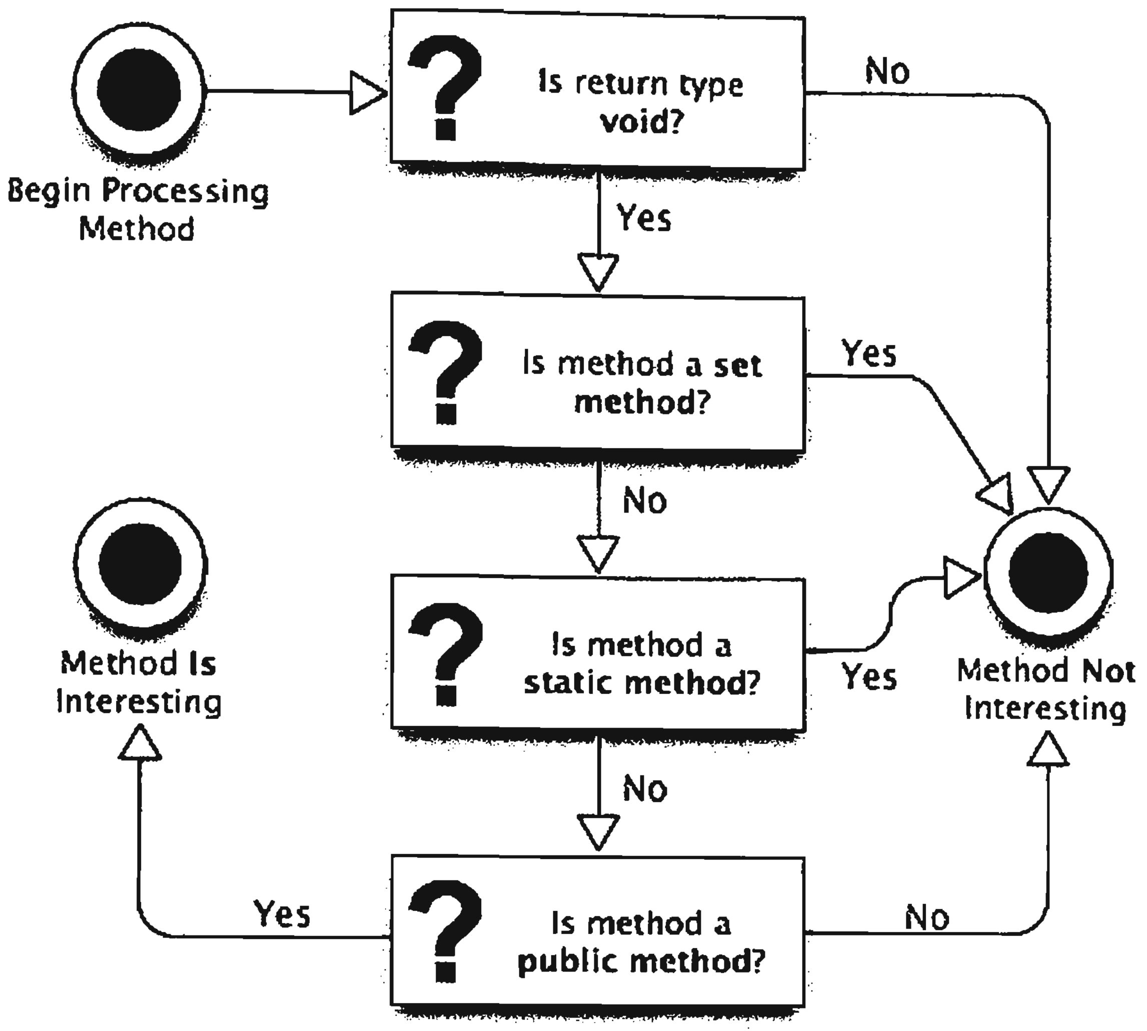


Figure 7-11: Method Suitability Assessment

As HLA interactions represent transient messages, there is no concept that can easily be mapped to a return type. Any method that does not have a void return type is immediately discounted as uninteresting. If the method begins with "set" it is adjudged to be following the common idiom of representing a mutator method[23]. In the HLA, strict ownership rules define who is and is not allowed to alter attribute values, and as such, mutator methods are also ruled uninteresting.

As there is no concept that can be directly mapped to class (static) data or methods in the HLA, these methods are also ruled out. Finally, non-public methods are not considered suitable for presence in the FOM as they designate methods whose access is restricted and suggest strongly that the method in question is used for internal model processing rather than providing some facility intended for use by other components.

[23] The specifics of how mutators are represented is dependent on the implementation platform. In the reference implementation, the `setXXX()` form is used. In other platforms, this would be replaced by the appropriate representation.

7.4 Rendering Configuration Data

The result of the processes introduced in sections 7.2 and 7.3 is a complete, in-memory, HLA object model. However, at this point there are two puzzle pieces that are needed to execute a Simspect powered HLA simulation that have not yet been created. Firstly, the FOM must exist in a federation description document that can be used by the RTI to define the model for a federation. Secondly, Simspect requires a mapping configuration data file so it knows what data to take action on and what to ignore.

The final step in the automatic generation process it the conversion of the in-memory object model into these necessary simulation artefacts. Figure 7-2 introduced the concept of a *Renderer*. It is the job of an individual Renderer to take the in-memory object model and generate some required artefact from it. The Somputer itself contains an arbitrarily sized collection of Renderers who are invoked before it exits. In the reference implementation, on of these produces the object model file in OMT format, while the other produced the Simspect XML configuration file. The exact number and purpose of Renderers is entirely dependent on the implementation environment in use.

7.5 Experiment Two

In section 5.2.3, the requirements for the second experiment were presented. The first experiment focused on validating the behaviour of the Simspect framework and the algorithms it used. However, when running the first experiment it was deemed acceptable for the FOM and Simspect configuration file to be hand written, thus allowing an experienced HLA developer to apply some semantic reasoning when deciding what parts of the pure-model should or should not be included in the HLA model. The purpose of this experiment is to validate that the automated process presented in this chapter constitutes a valid replacement for that manual process.

To assess whether or not this has take place, experiment one must be rerun with the exception being that no hand crafted articles can be used. Both the object model file and the Simspect configuration file must be automatically generated. Removing manual intervention from the process completes a workflow that allows an HLA simulation to be generated from a pure-model without the requirement of HLA expertise. The pure-OO code must be given to Simspect and it is expected that the Somputer and runtime environment handle all HLA needs.

Criteria One: Artefact Automatically Generated

This criteria is considered to be met by ensuring that no manually created deployment artefacts were used when running the experiment. The pure-OO code was first passed to the Simspect Compiler. It is the job of the compiler to weave the Simspect Aspect into the OO code, and now, to generate the necessary artefacts. Previously, a manually created FOM and mappings file were copied into the directory before execution. This time, the generated files were used.

As a further form of validation, the generated files can be inspected for differences that exist compared to the ones used in experiment one. The two generated files have been included in the supplementary materials that accompany this work. When inspecting them, evidence of the automation process can be seen in the areas of the files that deal with interaction parameter names. As noted earlier in this chapter, due to a limitation in the Java reflection framework, it is not possible to extract the names of parameters to a method. Consider the following:

```
(class Restaurant_printMe reliable timestamp
    (parameter param0) ;; type=Type.BOOLEAN
)
```

Listing 7-1: Generated Parameter Names for Methods

In the hand created FOM, the name of this parameter is known to the developer and is thus included in the file. However, this information is not available to the Simspect compiler, so the parameter is given the default name of "`param0`." The same situation occurs in the Simspect configuration file:

```
<method jmethod="testcode.racesim.Race.enterCars"
        hmethod="InteractionRoot.MethodCall.Race_enterCars">
    <param jposition="0" hname="param0" type="REFERENCE_ARRAY"/>
</method>
```

Listing 7-2: Generated Parameter Names for Methods

Another interesting difference exists in the generated FOM and configuration file. The compiler has identified the `printMe(boolean)` method of the `Restaurant` class as interesting. This is an example of a false-positive. This method meets all the functional requirements of a method as discussed in section 7.3, however, applying some semantic understanding of the purpose of this method leads us to determine it is not necessary for inclusion in the HLA object model (hence its absence from the model in experiment one). This highlights the inevitable limitation of the an automated process and is an excellent starting point for further work investigating how such instances could be reduced or eliminated.

Criteria Two: Pure-model must remain HLA free

As with experiment one, the pure-model code has not been edited in any way to allow this experiment to run. The code for the model is included in the supplementary package that accompanies this work to allow its inspection.

Criteria Three: Results of AOP-model do not match Pure-OO model

The final criteria is the same for experiment two as it was for experiment one. If all the internal operations are successful, then the results of the model that is run in an HLA federation with the companion federate should be different from those obtained when running the pure-OO model by itself. Should the model and the companion federate be interoperating correctly, the companion will effect the simulation and the final output generated should reflect this. The companion federate is the same one that was used in the previous experiment. Accordingly, the impact it has on the Race and Sushi simulations should be the same.

The Race Simulation

The following output was captures from the race simulation when run by itself and when run in an HLA federation with the companion federate:

Pure (non-AOP) Model:

```
Starting race...Race Over
[1]: Fastest Car      0:10:00
[2]: Medium Pace Car  0:15:01
[3]: Slowest Car      0:30:00
```

Distributed, AOP-Model:

```
Starting race...Race Over
[1]: RemoteCar        0:05:02
[2]: Fastest Car      0:10:00
[3]: Medium Pace Car  0:15:01
[4]: Slowest Car      0:30:00
```

The impact of the companion federate is evident by the presence of an additional car in the race.

The Sushi Simulation

Below are the results generated when running the Sushi simulation by itself and in an HLA federation with the companion federate:

Pure (non-AOP) Model:

```
sushi: INFO    Restaurant Simulation Over, No more food left!
sushi: INFO    Closing time: 30.0
sushi: INFO    Table Listing:
sushi: INFO     ->Table 1: customer=CustomerOne(5), availableDish=null
sushi: INFO     ->Table 2: customer=CustomerTwo(6), availableDish=null
sushi: INFO     ->Table 3: customer=CustomerThree(2), availableDish=null
sushi: INFO     ->Table 4: customer=null, availableDish=null
```

Distributed, AOP-Model:

```
sushi: INFO    Restaurant Simulation Over, No more food left!
sushi: INFO    Closing time: 26.0
sushi: INFO    Table Listing:
sushi: INFO     ->Table 4: customer=RemoteCustomer(7), availableDish=null
sushi: INFO     ->Table 1: customer=CustomerOne(0), availableDish=null
sushi: INFO     ->Table 2: customer=CustomerTwo(6), availableDish=null
sushi: INFO     ->Table 3: customer=CustomerThree(0), availableDish=null
```

The results once again show that the companion federate has an impact on the simulation through the inclusion of a customer that does not exist when running the model by itself. Further, this customer also consumes dishes, meaning the distribution across all other customers is altered.

As the companion federates are the same as used previously, these are the same results that were witnessed in experiment one. This demonstrates that the automatically generated object model and Simspect mappings file are valid. The companion federate is able to influence the final results in the same was it was when the object model and Simspect mappings information (representing the specification of which parts of the model are "interesting") were hand crafted. As the automated process removes the need for any HLA-specific knowledge to be used when converting the pure-OO model into an HLA aware version, the goal of this experiment is realised.

7.6 Summary

This chapter has presented a method for identifying data and methods within a pure-OO model that may be of interest in the context of an HLA simulation. The process used is entirely automated, thus removing any mandate for HLA experience of specialist knowledge, a major force driving this research. The second experiment demonstrated that the process produced all the necessary artefacts that allowed both test simulations to proceed as they did in the previous chapter when these items were hand crafted. The successful completion of this experiment demonstrates that the automatic generation of an HLA model from pure-OO code is possible using the techniques developed as part of this research. As a final form of validation, the next chapter describes a final experiment that seeks to combine an existing HLA simulation with pure-OO code.

Chapter 8

Air Transport Operations

The previous two chapters have discussed an environment and methodology for allowing pure-OO models to operate within HLA federations through an automated process. Experiments one and two have demonstrated the operation of these proposals and validated that they meet their goals as introduced in Chapter 5. This chapter presents the final experiment of this research: the integration of a pure-OO model into an existing HLA simulation, requiring it to interact fully with the other federates and co-operatively model its scenario.

8.1 The Air Transport Operations Simulation

The Air Transport Operations (ATO) simulation is a scenario used as a teaching aid in an HLA course offered by the University of Ballarat (UB) [119]. It describes a simulation that models Aircraft as they fly between various Airports. The three main federates that participate in the simulation are:

The **Aircraft Manager** (ACM) federate. The ACM is responsible for creating Aircraft objects and updating their state as they fly around the simulated environment. It sends and receives interactions when other federates need to control its actions. For example, when an Aircraft wants to land at an Airport, it sends a `RequestLand` interaction, and when the Airport is ready for it, a `Land` interaction is sent back.

The **Air Traffic Control** (ATC) federate is responsible for all the airports and associated Runways. It controls which planes can land at the various airports and when, potentially telling aircraft to loiter, divert or land.

The **Flight Manager** (FM) federate is responsible for deciding where each plane should fly to, how long it has to wait between flights and when maintenance is required. When a plane has landed, the FM issues it directions as to what to do next.

The implementation used in testing was completed by UB students and it used as part of a suite of test simulations for the Portico open source RTI. It includes a fourth federate that provides a GUI-based visualisation of that activities of the federation. This GUI, along with log file data will be used to determine whether or not the pure-OO model placed in the simulation is behaving appropriately.

8.2 Experiment Three

In section 5.2.3, the requirements and qualification of success for experiment three were introduced. To assess whether or not the experiment was successful, a log of the output generated by the pure-OO model was collected and a visual confirmation of the expected behaviour was captured through the visualisation tool provided with the ATO federation implementation.

The final experiment involves the existing ATC, ACM and FM federates operating in a federation with a pure-OO model. The role of the pure model is to create and manage a new airport. If successful, evidence that the FM federate is directing traffic to this new Airport and that the ACM is piloting Aircraft to the proper location should be found. The code for the pure-model is provided in the supplementary package that accompanies this thesis.

8.2.1 Results

Criteria One: OO-model runs without error

The first success criteria requires that the pure-OO model execute with the ATO federates without error. This is confirmed with a visual inspection of the execution process. Further, success in the second and third criteria imply the success of this one.

Criteria Two: ATO entity information discovered and used in OO-model

During execution, information created by the existing ATO federates must be made available to the pure-OO model. To ensure this is happening, some basic log statements are placed within the pure model. The capture below demonstrates this working:

```
[tim@pc-00244:simspect-1.0]$ java -cp out.jar testcode.ato.Freedonia
***** Register Freedonia Airport
***** Open the Airport!!!
awaitDepature(LLA114,FREEDONIA)
cleared(LLA114) destination=FREEDONIA
PINGED  Pinged aircraft [LLA114] from [FREEDONIA] (distance=1.9914730784283132)
requestLand(FREEDONIA, false, LLA114)
```

Listing 8-1: Freedonia OO-Model Log Output

In this listing, the "awaitDepature" and "cleared" entries come from the `Aircraft` class. The code in those methods just generates the logging statements you see. In the Simspect configuration file, those methods are mapped to the appropriate ATO FOM interaction classes. These captures show that the pure-model is receiving these interactions and that the appropriate methods are being called.

Criteria Three: Actions in OO-model affect simulation state

The final criteria for experiment three requires that actions within the pure-model have an affect on the broader simulation. In the previous listing, the "PINGED" entry was generated by the main method in the Freedonia class that controls the pure-model Airport. It continues in an loop, finding all the Aircraft that are close to it and sending them the Ping interaction. When an Aircraft gets this interaction, if the source is the Airport it wants to land at, it sends back a RequestLand interaction. In the pure-model, that method is mapped to the requestLand(String,boolean,String) method of the Airport class. The implementation for that method looks like this:

```
public void requestLand( String airportDesignator,
                         boolean emergency,
                         String aircraftDesignator )
{
    // only take action if the request was meant for us
    if( airportDesignator.equals("FREEDONIA") == false )
        return;

    System.out.println( "requestLand("+airportDesignator+", "+emergency+
                        ", "+aircraftDesignator+")" );

    // find the aircraft and let it know it can touch down
    Aircraft aircraft = Aircraft.findAircraft( aircraftDesignator );
    if( aircraft != null )
        aircraft.land( aircraftDesignator );
}
```

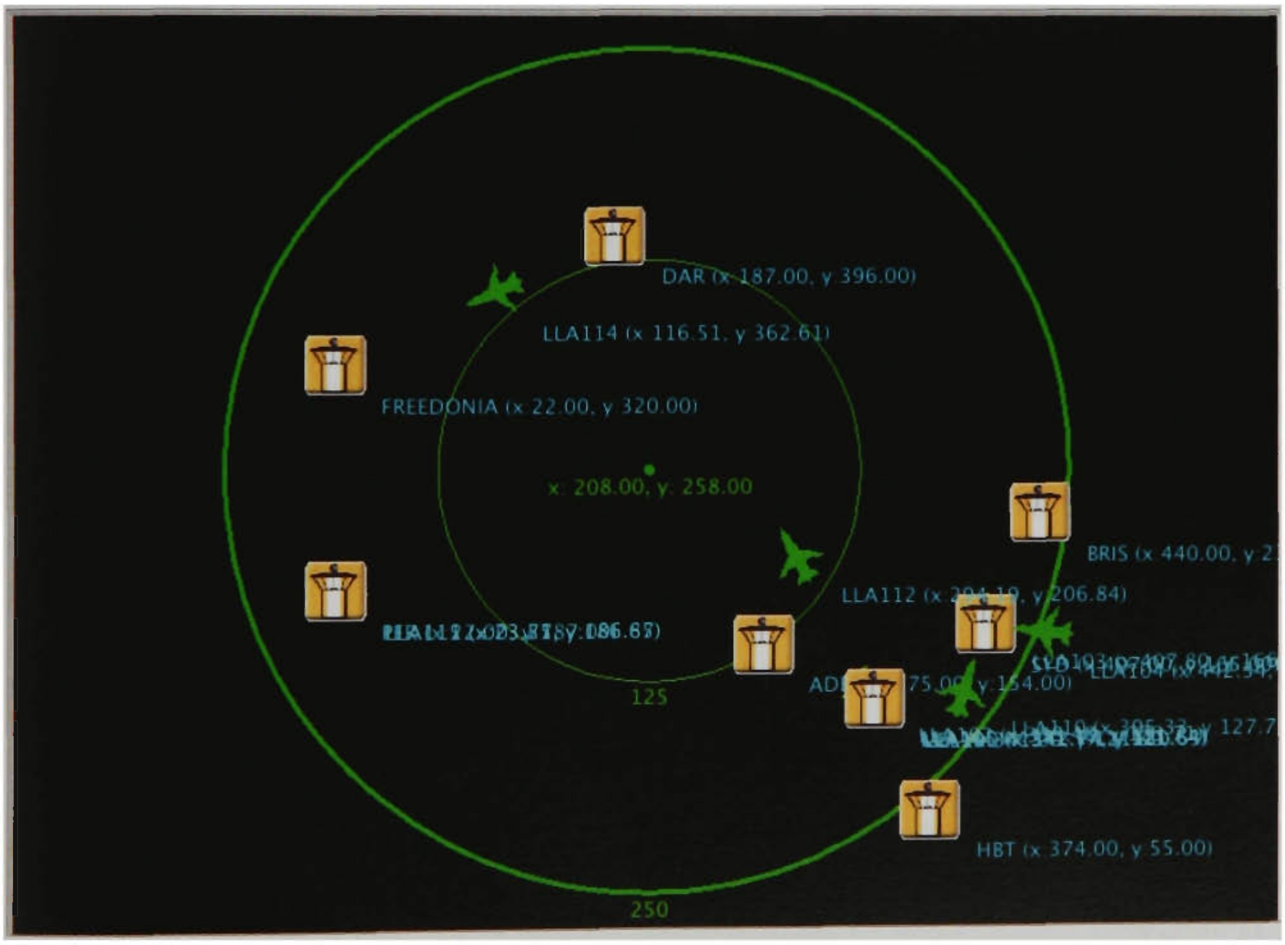
Listing 8-2: Request Land OO-Model Method

As you can see, the code in this method just locates the appropriate Aircraft instance and calls its land(String) method. For reference, the implementation of that method in the pure-model is as follows:

```
public void land( String aircraftDesignator )
{
    // Implemented by ACM federate. Calling this method triggers an interaction
    // that is responded to by the legacy HLA federate
}
```

Listing 8-3: Land OO-Model Method

Simspect captures this method call and turns it into the appropriate interaction, which is then sent out over the HLA and picked up by the ACM federate which begins landing at the Airport. This successfully shows the effect of an action internal to the pure-OO model being realised in the wider simulation. The following is a screen capture from the ATO federation visualisation utility.



This application is just another HLA federate and it shows the Freedonia airport being detected and displayed at the correction location, along with an Aircraft en-route to it. This demonstrates the successful completion of all requirements for the final experiment.

8.2.2 Remaining Problems

Although the final experiment is largely a success, it is not an unqualified one. A number of problems were encountered when attempting to successfully integrate a pure-OO model and an existing HLA simulation. This section discusses these issues.

Modification to Existing Simulation

One of the main problems with integrating any existing HLA simulation with pure-OO models developed using Simspect is the federate-level agreements Simspect itself imposes. In the ATO federation, these problems manifested themselves primarily in the space of interaction classes.

As discussed in Chapter 6, to align interactions with method calls, Simspect expects a specific interaction class hierarchy. In reality, the hierarchy can be done away with as all that is really important is that interactions come with a parameter that identifies the HLA object handle of the instance an interaction relates to; the `targetObject`. There are a some problems here.

Firstly, not all interactions are in reference to a particular HLA object. In these situations, the semantic disconnect between the purpose of HLA interactions and OO methods means that these scenarios cannot be supported. In the case of the ATO federation however, this is not a concern. All relevant interactions can are targeted at a specific instance. Interactions that ping an Aircraft, tell it to land, loiter, divert or await departure to a new destination are indeed all aimed at a particular `Aircraft` instance. Interactions that request a landing or take-off are aimed at a particular `Airport`. So while the expectations of Simspect may present problems in some situations, in such cases the problem is a product of HLA/OO conceptual misalignment that restricts Simspect's operation.

However, in the ATO implementation being used, the lack of a parameter specifying target object information was a problem. Although all interactions did have an identifying parameter, it was typically couched in terms of Aircraft or Airport designators (a string name). The Simspect reference implementation requires an object handle formatted as an integer. This is largely an implementation concern. Support could be added to the Simspect configurator to allow users to manually specify what parameter contains the target object information and how that information can be turned into an HLA object handle, but the path of least resistance in the experimental setting was to modify the existing FOM and simulation to include the object handle in all relevant interactions. This also necessitated manually tweaking the generated Simspect configuration file to map methods to the correct HLA interaction classes. As with the other experiments, the source code used in this experiment is available as part of the supplementary package that accompanies this work.

<u>Semantic Mismatch</u>

I have already presented one solution in which the semantic mismatch between OO and HLA has presented problems when integrating pure models and legacy simulations. However, these problems spread further. During the development of the pure-model, some "awkward" approaches were necessary to develop a functional component. For example, consider the `requestLand(String,boolean,String)` method in pure model code. It requires the manual conversion of an Aircraft designator into an `Aircraft` instance. In the OO world, a reference to the Aircraft would be passed directly to the method, but as an interaction is not meant to be a direct analogue to a method call, niceties such as this are not considered and the code must manually find the appropriate entity from a central store.

Another example of some awkward development comes in the form of direct data introductions. Unlike situations where pure-OO models are interacting with one another via the HLA (and thus can all impose the "OO way" on all simulation participants), data that is created in remote ATO federates must be manually introduced to the pure-model. In this experiment, the direct introductions facility described in 6.3.2. This speaks to the research question:

"How can pure models, that know nothing of application distribution, be created to depend on and work co-operatively with other remote models?"

To work co-operatively with the other federates, their remote data must be introduced into the model. However, the necessity of direct introductions places requirements on the way the pure-model must finds and stores its data. Where more sophisticated approaches could be used in the other experiments (such as encapsulating all `Cars` within a `Race` instance, that itself has additional information), when interfacing with a legacy HLA model, the full benefits of OO are not available. Although annoying, this flaw is not fatal, and workable solutions can indeed be developed, as the success of experiment three stands in testament to.

<u>HLA Knowledge Not Entirely Eliminated</u>

As one can already understand from the previous subsections, when integrating a pure-model with an existing HLA simulation, it does not appear likely that one totally eliminate the need for HLA knowledge. Federate-level agreements that are different for each and every simulation must be dealt with. This brings to mind one of the research questions this experiment was meant to address:

"Can the definition of federate level agreements be expressed without requiring manual intervention?"

Without any formally specified, machine interpretable standard for specifying federate level agreements such as the problem mentioned above, the HLA lacks the vocabulary to express federate-level agreements, and as such restricts any attempt to define a generic solution. It is impossible to develop a solution for a situation whose requirements are unknown. Federate level agreements can only be solved with manual, human interpretation and intervention.

Another example that arose during experiment three was how the pure-OO model would fit into the ATO federation with regard to execution requirements. The ATO federation expects all federates to follow a specific sequence of synchronization points, performing predefines actions at each one. To deal with this facility, a custom execution manager had to be developed and used. Simspect supports the ability to dynamically specify a class that encapsulates such requirements, but the development of such a component again requires HLA knowledge.

8.3 Conclusion

Overall, the final experiment can be considered a success. A pure-OO model was able to co-operatively interact with an existing HLA federation with only minimal intervention. Although this does not meet the full goals of this research (to eliminate the requirement of HLA knowledge entirely), it is reasonable to expect that when interfacing with a legacy HLA simulation, that some form of HLA knowledge be available. Having addressed the final research questions, the next chapter will conclude with a discussion of what this research has achieved and flag some areas that have strong potential investigation.

Chapter 9

Conclusion

As discussed in the early chapters of this thesis, a vast range and number of tools are used for simulation purposes. From custom-built tools specialising in helping users compose simulations for specific scenarios, to general purpose tools such as spreadsheets. While each type of tool used for simulation in the wider business community presents advantages and disadvantages, a common shortcoming among all is a lack of interoperability. In a setting where numerous differing tools are all used for similar purposes, the ability to leverage investments made in the development of simulation models, independent of those tools, is severely restricted.

Distributed simulation brings with it many benefits. From providing an environment in which interoperability can be increased to allowing larger and more complex scenarios to be played out, there is much to gain from allowing existing tools to leverage distributed simulation technologies. However the prohibitive costs that are associated with distributed simulation and the limited supply of these skills puts the potential benefits of distributed simulation technologies beyond the reach of those using commodity tools for simulation purposes. A solution to this particular problem is considerably attractive and has the potential to enable far richer analysis in such an environment and to increase the overall usefulness of both these tools and simulation in general.

However, the sheer volume of different tools used within the wider business community restricts one's ability to define a solution that can readily be slotted into them all. Each tool has its own particular way of representing models and its own way of interfacing with its underlying simulation library. In such a heterogeneous environment, locating or defining a general conceptual solution that could possibly satisfy all approaches becomes practically impossible. Chapters 2 and 3 discussed this problem, proposing that at some level, Object-Oriented programming would form a lowest common denominator for the largest number of environments. A solution at this level would provide something that has the potential to be readily integrated into many simulation tools, both those existing and yet to be developed.

Interfaces to distributed simulation technologies exist in the most common OO programming languages, but leveraging these technologies introduces the requirement of possessing specialised skills, which in this research comes in the form of HLA understanding. The barrier to entry presented by the HLA is steep, and although Object

Oriented programming skills are mainstream and widespread, any solution that revolves around this community gaining a full and working understanding of the HLA is unreasonable.

This thesis proposes that the development of a solution that could allow pure-OO models to be automatically rendered as HLA components would form a significant contribution towards making distributed simulation more readily accessible in domains where its use thus far has been negligible. Identifying and solving the problems from the level of OO through the HLA advances the current state of the art and presents a solution that can be readily integrated into existing simulation tools, opening the gate to a richer simulation ecosystem.

Chapter 4 highlights some potential solutions that could be leveraged in pursuit of this goal. Of these, two primary candidates stood out in the literature: The Model Driven Architecture (MDA) and Aspect-Oriented Programming (AOP). Although initially appealing because it proposes to allow users to express their problems in their own terms, the MDA was discounted due to numerous practical problems preventing the full realisation of its ambitious intent. Conversely, while not as grand in vision as the MDA, AOP is a solution for which many practical implementations already exists and which supports mission critical services in a number of domains. Allowing platform specific concerns such as the HLA to be separated and isolated from the development of other parts of a system, AOP was demonstrated to be the most promising path.

AOP alone does not totally solve the problems this research seeks to address. While the HLA portion of a system can be quarantined, it must still be developed. Further, someone must identify the points within a model that are of interest in the HLA context and describe how the OO-model maps onto the HLA-model. Each of these activities require a high degree of competence with the HLA and associated technologies. AOP does not eliminate the need for HLA expertise, it just contains it.

The focus of this research is on that particular gap, proposing an environment that leverages AOP to reduce the scope of HLA concerns and then presenting methods for automatically extracting the intent of an OO-model as it relates to the HLA. The Simspect framework is the realisation of these goals, comprising an AOP-based environment and methods for identifying and extracting pieces of an OO-model that might be of interest in an HLA context.

Chapter 6 described how an AOP framework could successfully isolate HLA logic, allowing a model to be developed and described in pure-OO. The activities of this model were then intercepted at runtime and selectively passed to the HLA based on manually identified mapping points. The first experiment validated that this framework met its goals, but still required manual intervention which necessitated HLA knowledge. This advancement demonstrated how the proposed concept was technically valid, moving the focus to methods for introspecting OO models in an effort to automatically extract information on interest when connecting with the HLA.

The second experiment looked at methods for identifying the relevant information within an OO model, allowing the manual processes of generating an HLA object model and identifying and defining the various mapping points to be automated. Chapter 7 discussed and validated these approaches, demonstrating the process of automatically developing an HLA federation from pure-OO code, allowing the previously manual processes to be removed. This achievement is central to the goals of this research. The value that such a facility presents represents a significant contribution to the current state of the art and acts as a facilitator, bringing the benefits of distributed simulation within reach of commodity tools.

Further Work

The final experiment involved the integration of a newly developed OO-model with an existing HLA simulation. While successful, this chapter highlighted some of the challenges that remain when considering the underlying semantic disconnect that exists between OO and the HLA. Throughout this thesis, a number of areas that could be considered fertile grounds for further exploration have been identified. Primary among these is exploring ways to specify federate-level agreements such that they can be captured in an automated fashion or used without necessitating a deep understanding of esoteric HLA details.

Unfortunately, the HLA standard as it currently exists allows federations significant enough flexibility in the definition of these details that it becomes extremely difficult to develop a generically applicable solution. Although this flexibility is a great benefit to HLA developers, the lack of any solution to codify these agreements in explicit enough detail (and in a machine readable manner) restricts any solution. Further investigation into ways to solve this problem would help allow the methods presented in this thesis to become even more effective when integrating OO and HLA models.

A second area where more research would be beneficial is in ways to extend the proposed framework to cover the complex-type enhancements that were introduced in the IEEE 1516 version of the HLA standard. Although the use of these types does goes against some of the philosophical benefits of selective interest in an object model, their use has become quite prevalent. Questions about how best to identify types within an OO-model that would be best suited to complex types rather than HLA object classes would be an interesting study and one that would allow the integration of this work with even more existing simulations.

Finally, the work presented in this thesis can aid in the integration of modelling and simulation functionality into existing operational systems used within the military. Facilitating an easier transition into the network-centric world of distributed simulation helps to gain further benefit from these systems by allowing them to be used not only for operational purposes, but also more directly in simulated training exercises.

Summary

Through the proposed framework and methods presented in this research, a solution that has the potential to significantly reduce the cost and knowledge required to leverage distributed simulation technologies in commodity tools often used for simulation in the wider business community has been achieved. This step forward places within reach of these tools the potential for a much richer tool set allowing more complex and deeper analysis to be produced, which could in turn provide those who rely on such information a greater understanding of the effects their decisions will have.

Through the proposed framework and validated in experimentation, this research presents a solution that has the potential to significantly reduce the effort and knowledge required when developing HLA models, making access to the HLA, and the benefits distributed simulation provides, available to a much broader audience and able to be more readily integrated into the tools people commonly use for simulation purposes.

Appendix A

The Race Car Simulation

The Race Car simulation is designed to be a structurally simple pure-OO simulation. It contains no inheritance hierarchies and only minimal aggregation and composition. It is the first barrier used when assessing any solution that proposes to allow the automatic rendering of OO-models as HLA simulation components, ensuring that at least the basics are working.

The class structure of the Race simulation is shown in Figure A-1:

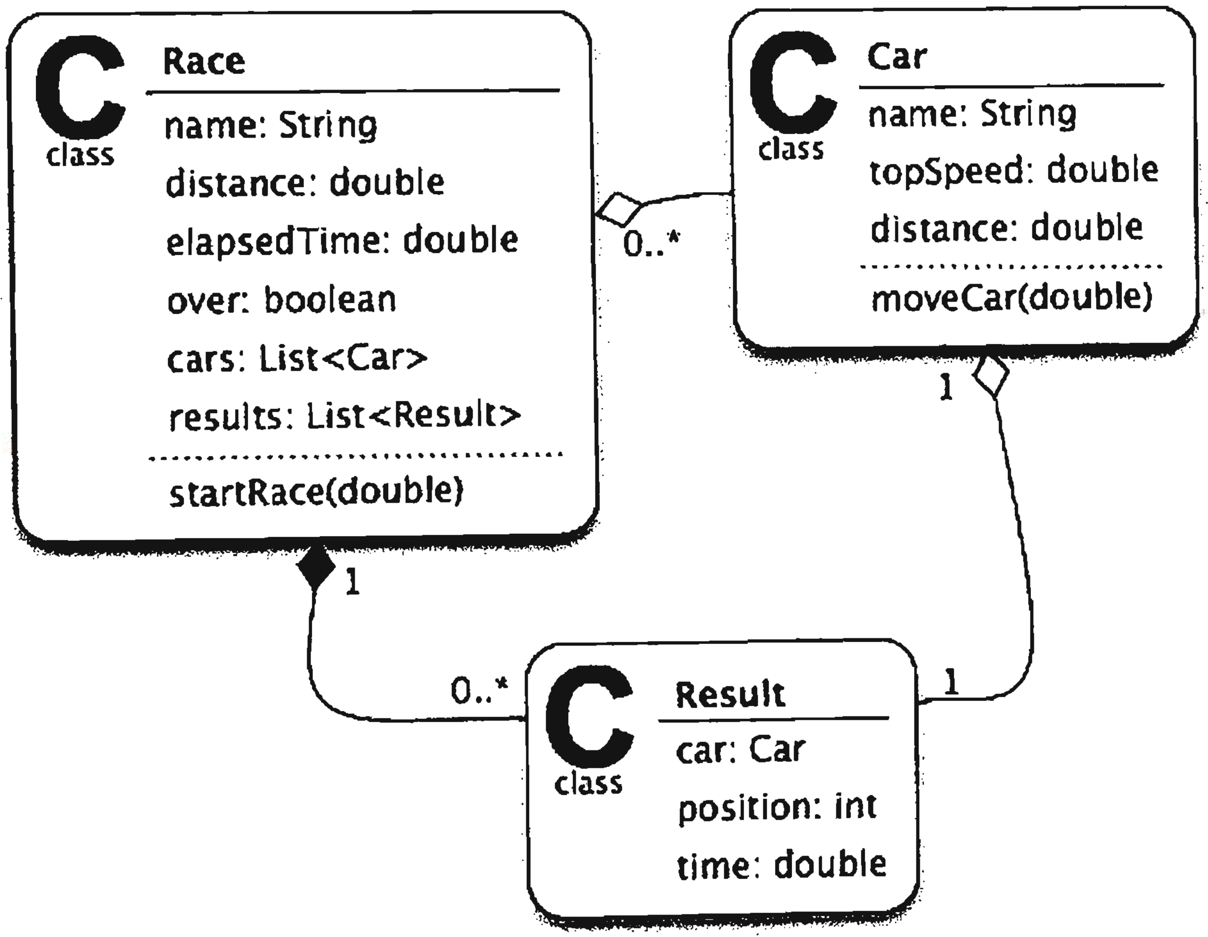


Figure A-1: Race Car Simulation Class Diagram

The simulation has a single Race, inside which any number of Cars can be present. When the simulation starts up, Car objects are manually instantiated and entered into the race (three of them in the case of the experiments used in this research). The race is then started via a call to `startRace(double)`. This method loops over each of the Cars in the race, continually calling their `moveCar(double)` methods. The original parameter to `startRace()` specifies the increments time should be advanced in.

To keep things predictable, thus making experimentation easier, the model used to advance Car objects through the race is simple. As soon as the race starts, each Car is assumed to be travelling at its maximum possible speed. Therefore, the car with the highest top speed will always win. The Race iterates over each Car, asking it to advance itself a little bit further in time each iteration. When the `distance` of the Car surpasses the `distance` of the race, a new `Result` object is created, recording the position of the Car with regard to the field and the current elapsed time. Once all Cars have passed the finish line, the race is over and the results are printed.

The Race Car simulation was designed to support experimentation by allowing remote components to insert new cars into a race and to control their advancement according to whatever model the remote entity chose.

Appendix B

The Sushi-Boat Simulation

The Sushi-Boat simulation is designed to be more structurally complex than the Race Car simulation. It contains many common object-orientation constructs such as abstract classes and inheritance trees of reasonable depth. It also contains considerable levels of aggregation and composition, more accurately representing a real-world model.

Figure B-1 shows the majority of the classes involved in the pure-OO sushi simulation. In each simulation, there is a single `Restaurant`. Inside each `Restaurant` are a number of `Tables` at which `Customers` can be seated. The model is meant to simulate a sushi-boat or sushi-train style restaurant, where dishes continually revolve past each table, at which point the occupants can choose to take and consume the dish, or let it go. The `run()` method of the `Driver` class performs the necessary logic to move Dishes from one table to the next (which is done purely through accessor and mutator methods).

When a dish arrives at an occupied table, the resident `Customer` is notified through the `newDishHasArrived(Dish)` method. At this point, they can choose to consume the dish by calling its `eat(Customer)` method (passing themselves as the Customer), or it can ignore the dish. When a `Dish` is eaten, it is added to the Customers `dishesPurchased` set so it can be tallied at the end. The Driver continues to move dishes around the restaurant until they have all been consumed, at which point the simulation ends and information is printed describing the dishes and the valuation of the dishes each customer consumed.

The Sushi-Boat simulation is designed to support experimentation by allowing remote components to act as Customers, specifying their own algorithm for how food is chosen. The `PredictableCustomer` implementation is also used for experimentation, as it consumes dishes in a repeatable way. Placing a remote Customer into the simulation will affect the flow of dishes to the other customers, thus altering the final distribution of dishes.

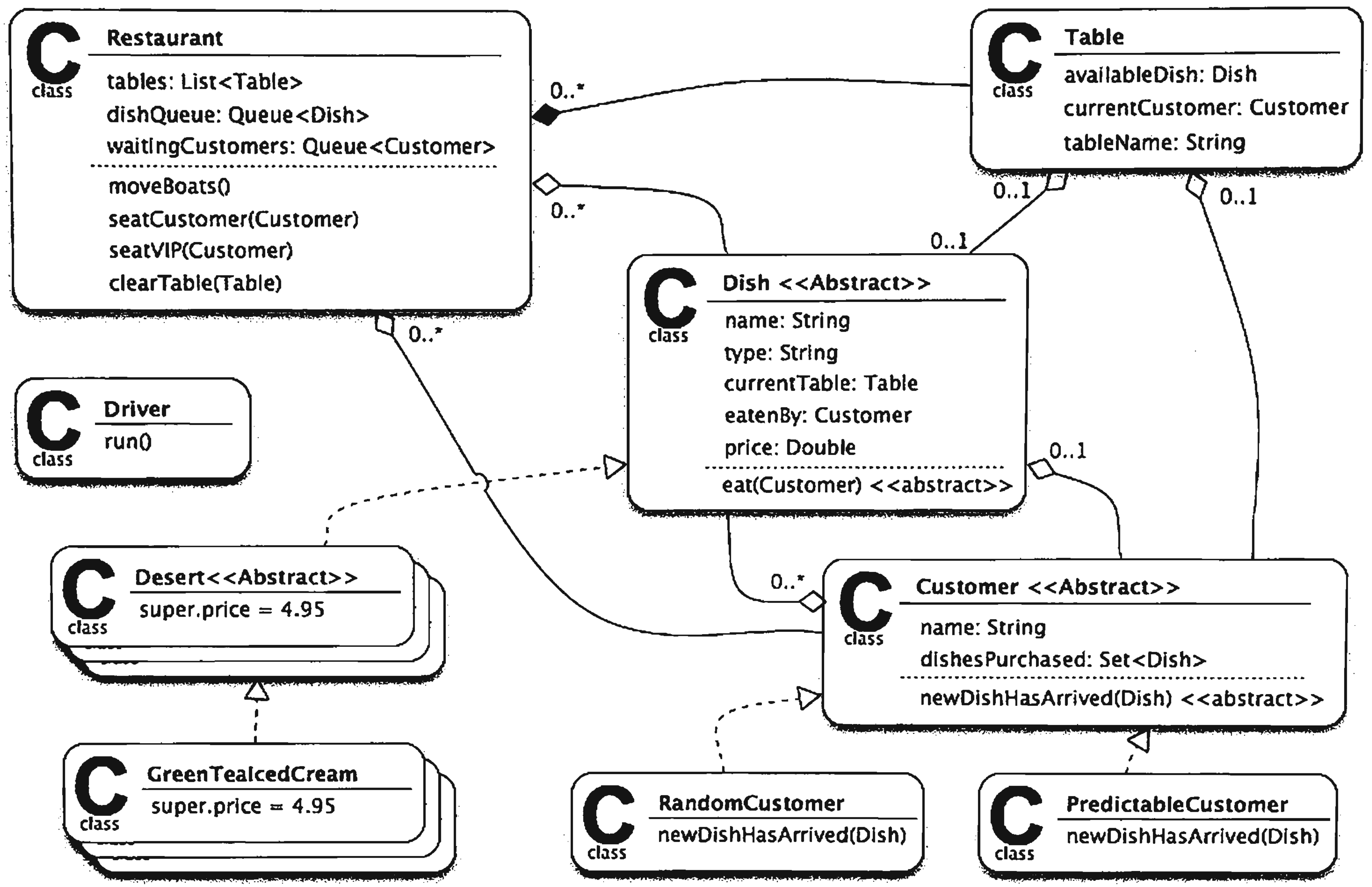


Figure B-1: Sushi Simulation Class Diagram

Appendix C

The Air Transport Operations Federation

The Air Transport operations is a simulation scenario used during practical exercises as part of a commercial 2-week course offered by the University of Ballarat. The basic premiss of the scenario is that it models the operations of a number of Aircraft as they transit between a groups of Airports.

The listing below describes the main structure of the FOM used in the ATO federation, starting with the object classes:

```
;; Object Classes
(class ObjectRoot
  (attribute privilegeToDelete)
  (class Position
    (attribute x)
    (attribute y)
    (attribute altitude)
    (class Airport
      (attribute designator)
    )
    (class Runway
      (attribute airportDesignator)
    )
    (class Aircraft
      (attribute designator)
      (attribute model)
      (attribute state)
      (attribute destinationAirport)
      (attribute currentAirport)
      (attribute groundSpeed)
    )
  )
)

```

Listing C-1: Air Transport Operations Object Model Classes

and the interaction classes:

```
;; Interaction Classes
(class InteractionRoot
  (class Aircraft
    (parameter targetObject)
    (parameter aircraftDesignator)
    (class AwaitDeparture
      (parameter destination)
    )
    (class TakeOff)
    (class Cleared)
    (class Ping
      (parameter airportDesignator)
    )
    (class Loiter)
    (class Divert
      (parameter reason)
    )
    (class Land)
    (class Landed)
    (class Turnaround)
    (class Repair)
    (class Dead)
  )
  (class Airport
    (parameter targetObject)
    (parameter airportDesignator)
    (class RequestLand
      (parameter emergency)
      (parameter aircraftDesignator)
    )
  )
)
```

Listing C-2: Air Transport Operations Interaction Model Classes

Three main federates make up the core of the simulation.

The **Air Traffic Controller** manages the various `Airports` that inhabit the simulation. This federate is responsible for sending ping notifications to `Aircraft` as they come into range of the airport and for handling takeoff and landing requests from the Aircraft in an orderly fashion.

The **Aircraft Manager** handles the take-off, in-flight and landing operations of a number of `Aircraft`. This federate is primarily responsible for moving the plane from one point to another, and controlling takeoff and landing.

The **Flight Manager** is responsible for Aircraft after they land, putting them in for maintenance and deciding when they are ready to be put into action again and where their next destination is.

Unlike the previous two models, the ATO is not designed with the intention of supporting experimentation by allowing extension in some pre-considered manner. This simulation exists as a legacy HLA model, the implementation of which was developed by the Author and student Michael Fraser during their time at the University of Ballarat.

There are a number of ways that the ATO federation could be extended:

- An OO-model could be designed to represent a particular Airport whose algorithm for deciding who can land, and when, it different from the first-come, first-serve style of the legacy implementation.

- An OO-model could be designed to represent an Aircraft, modelling its path and motion as it travels between airports.

- An OO-model could be designed to replace the Flight Manager federate, controlling how long an Aircraft may remain operational before maintenance is needed and the order of Airports on its route.

For experimentation in this thesis, the first option was chosen. A pure-OO model instantiates an Airport and has it automatically entered into the simulation. It then performs all the operations for that Airport that the ATC federate performs for the other airports. The Flight Manager sees it when it is registered with the simulation and is able to direct Aircraft to it.

The figure below shows the object model that was created and fed into Simspect:

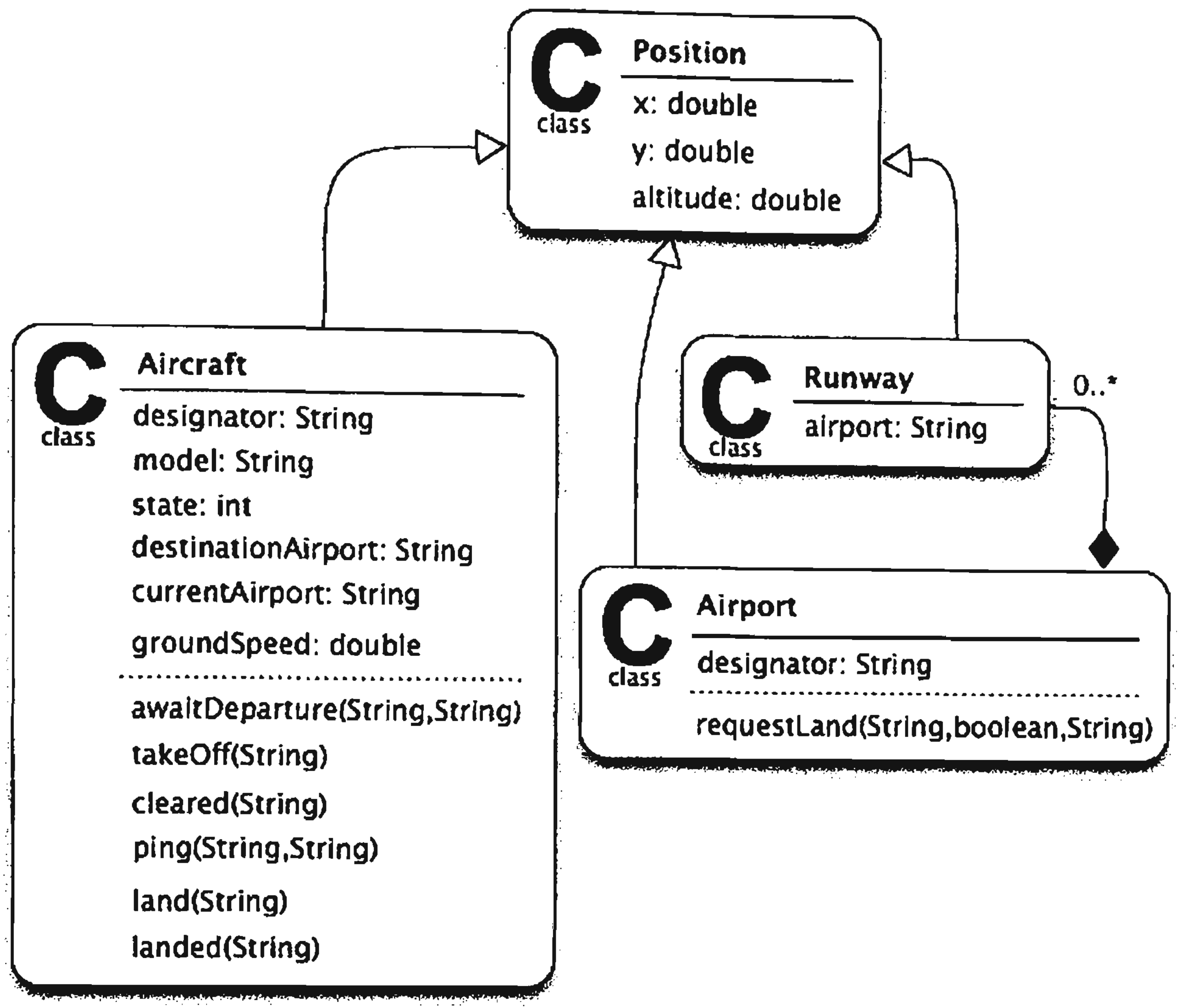


Figure C-1: Air Transport Operations Class Diagram

Running this application with Simspect requires the custom tweaking of the generated mappings configuration file. The primary reason for this is that the pure-OO model must interact with an unchangeable legacy HLA simulation. When integrating in this way, some manual mappings work, requiring HLA knowledge, can be considered reasonable. If a user is integrating with an HLA simulation it is fair to expect that some HLA expertise exists and that can be put to minimal use by ensuring the mappings are correct. This knowledge however is not needed when actually developing the pure model.

The pure model works by having a main simulate method call methods on the local `Aircraft` instances. Simspect sees that these are not local objects and triggers the mapped interactions to be sent. The same pattern is used in reverse for incoming interactions which in turn trigger methods on the local Airport instances. The pure-model implements the bodies of these methods as appropriate.

A fourth legacy federate was also used in experimentation. This federate acts as a visualisation of the simulation, allowing confirmation that the pure-OO model's Aircraft was entering the simulation and that it was moving from Airport to Airport.

Appendix D

Generic Weaving Rules and Advice

Below is a listing of the code for the Generic Aspect. Note that some code (such as that used for logging) has been omitted in the interest of brevity.

```
/*
 *   Copyright 2007 Distributed Simulation Lab, University of Ballarat
 *
 *   This file is part of simspect.
 *   For license details, see the LICENSE file.
 */
package simspect.aspects;

import java.lang.reflect.Field;
import java.lang.reflect.Method;

import org.apache.log4j.Logger;
import org.aspectj.lang.reflect.MethodSignature;
import org.aspectj.runtime.reflect.FieldSignatureImpl;

import simspect.runtime.ModelFacade;

public aspect SimspectExtractor
{
	//----------------------------------------------------------------
	//                     INSTANCE VARIABLES
	//----------------------------------------------------------------
	private ModelFacade facade;
	private Logger logger;

	//----------------------------------------------------------------
	//                        CONSTRUCTORS
	//----------------------------------------------------------------

	public SimspectExtractor()
	{}

	//----------------------------------------------------------------
	//                         POINTCUTS
	//----------------------------------------------------------------

	protected pointcut ignoreList() :
		!within( hla..* ) &&
		!within( simspect..* ) &&
		!within( com.lbf..* ) &&
		!within( org..* ) &&
		!within( testcode.exp..* );
		//!within( testcode..* );

	/** pointcut to get the main method */
	protected pointcut mainMethod() :
		execution( public static void main(String[]) ) &&
		ignoreList();
```

```
/** pointcut to get all consturctors */
protected pointcut constructors( Object newObject ) :
      initialization( public *.new(..) ) &&
      ignoreList() &&
      target( newObject );

/** pointcut to get all field sets */
protected pointcut fieldSet( Object target, Object newValue ) :
      set( * *.* ) &&
      ignoreList() &&
      args(newValue) &&
      target(target);

/** pointcut to capture all methods */
protected pointcut methodCall( Object target ) :
      execution( public !static void *.*(..) ) &&
      !execution( public * *.get*() ) &&
      !execution( public * *.set*(..) ) &&
      ignoreList() &&
      target( target );

//-----------------------------------------------------------
//                        ADVICE
//-----------------------------------------------------------
////////////////////////////
// CAPTURE: main() method //
////////////////////////////
// Capture the main method so that we can instantiate the runtime
void around() : mainMethod()
{
      try
      {
            // create the facade and runtime //
            this.facade = new ModelFacade();
            this.logger = this.facade.getLogger();

            // inform the facade of simulation beginning //
            this.facade.onStartup();

            // proceed and execute the model main method
            proceed();

            // tell the facade that things are done for
            this.facade.onShutdown();
      }
      catch( Exception e )
      {
            e.printStackTrace();
            System.exit( 1 );
      }

      // make sure that we get out of here //
      System.exit( 0 );
}

///////////////////////////
// CAPTURE: constructors //
///////////////////////////
```

```
before( Object newObject ) : constructors( newObject )
{
	// notify the runtime //
	this.facade.onConstructor( newObject );
}

/////////////////////////////
// CAPTURE: field SET call //
/////////////////////////////
Object around( Object target, Object newValue ) :
	fieldSet( target, newValue )
{
	try
	{
		// 1. determine which field is being set //
		String name =
		  ((FieldSignatureImpl)
		  thisJoinPoint.getSignature()).getName();
		Field field = getField( name, target.getClass() );

		// 2. notify the runtime //
		facade.onFieldSet( field, target, newValue );

		// 3. proceed with the execution //
		return proceed( target, newValue );
	}
	catch( NoSuchFieldException nsfe )
	{
		throw new RuntimeException( nsfe );
	}
}

//////////////////////////
// CAPTURE: method call //
//////////////////////////
Object around( Object target ) : methodCall( target )
{
	// 1. collect the necessary information //
	Method method =
	  ((MethodSignature)
	  thisJoinPoint.getSignature()).getMethod();
	Object[] args = thisJoinPoint.getArgs();

	// 2. notify the runtime //
	if( facade.onMethodCall(method, target, args) )
	{
		// proceed as normal //
		return proceed( target );
	}
	else
	{
		// we do not own the object the method is being called
		// on, skip the proceed()
		return null;
	}
}

//--------------------------------------------------------------------------
```

```
	//                          INSTANCE METHODS
	//------------------------------------------------------------------
	private Field getField( String name, Class<?> clazz )
		throws NoSuchFieldException
	{
		try
		{
			return clazz.getDeclaredField( name );
		}
		catch( NoSuchFieldException nsfe )
		{
			if( clazz.getSuperclass() != null )
				return getField(name, clazz.getSuperclass());
			else
				throw nsfe;
		}
	}
}
```

References

[1] Adelantado, M., Bussenot, J., Rousselot, J., Siron, P. & Betoule, M. (2004) *Towards a high Performance, high Availability Open Source RTI for Composable Simulations* In Proceedings of Fall Simulation Interoperability Workshop. Workshop paper: 04F-SIW-014

[2] Akkai, F., Bader, A. & Elrad, T. (2001) *Dynamic weaving for building reconfigurable software systems* In Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems

[3] Allen, R., Garlan, D. & Ivers, J. (1998) *Formal Modeling and Analysis of the HLA Component Integration Standard* In Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering. p70-79

[4] Ambler, A. L., Burnett, M. M., & Zimmerman, B. A. (1992) Operational versus definitional: A perspective on programming paradigm. *IEEE Computer 25* (9), 28-43

[5] Aspect J. Project Website. Last Retrieved on January 5, 2009 from: http://www.eclipse.org/aspectj/

[6] Atkinson, K. (2003) *Applying Enterprise Architecture to Modeling and Simulation* In Proceedings of Spring Simulation Interoperability Workshop. Workshop Paper: 03S-SIW-082

[7] Balci, O., Bertelrud, A., Esterbrook, C. & Nance, R. (1998) *Visual Simulation Environment* In Proceedings of 30th Winter Simulation Conference. 279-287.

[8] Banks, J. (1999) *Introduction to Simulation* In Proceedings of the 31st conference on Winder Simulation. 7-13

[9] Banks, J., Carson, J., Nelson, B. & Nicol, D. (2000) *Discrete Event System Simulation.* New Jersey: Prentice-Hall.

[10] Best, J., Canney, S. & Cramp, A. (2002) *Virtual Maritime System Architecture* In Proceedings of Fall Simulation Interoperability Workshop. Paper: 02F-SIW-102

[11] Best, J. & Cramp, A. (2002) *Execution Management in the Virtual Maratime Systems Architecture.* Technical Manual. Defence Science and Technology Organisation.

[12] Brown, P. & Gould, J. (1987) Experimental study of people creating spreadsheets. *ACM Transactions of Information Systems* 5 (3), 258-272.

[13] Burke B. & Brock, A. (2003) *Aspect-Oriented Programming and JBoss.* Last Retrieved on January 5, 2009 from: http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html

[14] Buss, A. & Jackson, L. (1998) *Distributed Simulation Modeling: A Comparision Of HLA, CORBA And RMI* In proceedings of Winter Simulation Conference. 819-825.

[15] Butler, R. (2000) *Is This Spreadsheet a Tax Evader?* In Proceedings of 33rd Hawaii International Conference on System Sciences. Retrieved from: http://panko.cba.hawaii.edu/ssr/HICSS33/HICSS33-Butler-Evader.pdf

[16] Calytrix Technologies, *FOM Development.* Last Retrieved on January 5, 2009 from: http://www.calytrix.com/siteContent/SIMplicity/FOM.php

[17] Calytrix Technologies, *Creating Federates.* Last Retrieved on January 5, 2009 from: http://www.calytrix.com/siteContent/SIMplicity/federates.php

[18] Calytrix Technologies, *Mappings: Transformation and Toolkit Operations.* Last Retrieved on January 5, 2009 from: http://www.calytrix.com/siteContent/SIMplicity/Mappings.php

[19] Calytrix Technologies, *Deployment Management and Initialization Data.* Last Retrieved on January 5, 2009 from: http://www.calytrix.com/siteContent/SIMplicity/DMID.php

[20] Cazard, L. (2002) *HLA Federates Design and Federations Management: Towards a High Level Object-Oriented Architecture Hiding the HLA Services* In Proceedings of Spring Simulation Interoperability Workshop. Workshop Paper: 02S-SIW-013

[21] Ceccato, M. & Tonella, P. (2004) *Adding Distribution to Existing Applications by Means of Aspect Oriented Programming* In Proceedings of 4th IEEE International Workshop on Source Code Analysis and Manipulation. 107-116.

[22] Chwif, L., Barretto, M. & Saliby, E. (2002) *Supply chain analysis: supply chain analysis: spreadsheet or simulation?* In Proceedings of 34th Winter Simulation Conference. 59-66.

[23] Clarke, S. & Baniassad, E. (2005) *Aspect-Oriented Analysis and Design: The Theme Approach.* Addison Wesley: New York.

[24] Constantinides, C. & Skotiniotis, T. (2002) *Reasoning About a Classification of Crosscutting Concerns in Object-Oriented Systems* In Proceedings of Second Workshop on Aspect-Oriented Software Development. 1-6.

[25] CppReflect. Project Website. Last Retrieved on January 5, 2009 from: http://sourceforge.net/projects/cppreflect/

[26] Dahmann, J., Kuhl, F. & Weatherly, R. (1999) *Creating Computer Simulation Systems – An Introduction to the High Level Architecture.* New Jersey: Prentice Hall.

[27] Defense Modeling and Simulation Office (2001) *RTI 1.3-Next Generation Programmer's Guide Version 5.* Technical Manual.

[28] Deitel, H. J. & Deitel, P. J. (2007) *Java How to Program (7th Edition)*, Prentice-Hall: New Jersey

[29] Ditlea, S. (1987) Spreadsheets can be hazardous to your health. *Personal Computing,* 11 (1), 60-69.

[30] Elrad, T., Filman, R. & Bader, A. (2001) Aspect-oriented programming: Introduction. *Communications of the ACM*, 44 (10), 29-32

[31] Erwig, M., Abraham, R., Cooperstein, I. & Kollmansberger, S. (2005) *Automatic generation and maintenance of correct spreadsheets* In Proceedings of 27th international conference on Software engineering. 136-145.

[32] ExtendSim. Project Website. Last Retrieved on January 5, 2009 from: http://www.extendsim.com/

[33] Filman, R., Elrad, T., Clarke, S. & Aksit, M. (2004) *Aspect-Oriented Software Development.* Addison Wesley: New York.

[34] Filman, R. & Friedman, D. (2000) *Aspect-Oriented Programming is Quantification and Obliviousness* In Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems.

[35] Forman, I. & Forman, N. (2005) *Java Reflection In Action.* Manning Publications: Greenwich.

[36] Fowler, M. *UML as Sketch.* Last retrieved on January 5, 2009 from: http://martinfowler.com/bliki/UmlAsSketch.html

[37] Fujimoto, R. (2000) Time Management in the DoD High Level Architecture. *ACM Transactions on Modeling and Computer Simulation* 10 (3), 268-294

[38] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software.* New York: Addison Wesley

[39] Garlan, D. (1995) Architectural mismatch: Why reuse is so hard. *IEEE Software* 12 (6), 17-26

[40] Granowetter, L. (1999) *Solving the FOM-Independence Problem* In Proceedings of Simulation Technology and Training Conference.

[41] Granowetter, L. (2003) *IEEE 1516 Compliance -- Will the Real C++ API Please Stand Up?* Technical Whitepaper. Last Retrieved on January 5, 2009 from: http://www.mak.com/pdfs/wp_1516_api.pdf

[42] Granowetter, L. (2003) *RTI Interoperability Issues – API Standards, Wire Standards, and RTI Bridges* In Proceedings of Spring Simulation Interoperability Workshop. Workshop Paper: 03S-SIW-063

[43] Granowetter, L. (2004) *Design of the Dynamic-Link-Compatible C++ RTI API for IEEE 1516* In Proceedings of Fall Simulation Interoperability Workshop. Workshop paper: 04F-SIW-086.

[44] Griss, M. (1993) Software Reuse: From library to factory. *IBM Systems Journal* 32 (4), 548-566

[45] Gustavson, P., Morse, K., Lutz, R. & Reichenthal, S. (2004) Applying Design Patterns for Enabling Simulation Interoperability. *Modelling and Simulation,* 3 (2).

[46] Heineman, G., Councill, W. (1998) *Component-Based Software Engineering.* Massachusetts: Addison Wesley

[47] Hlupic, V., Walker, P. & Irani, Z. (1998) Predicting movements in foreign current rates using simulation modelling. *Management Decision* 36 (7). 465

[48] Institute of Electrical and Electronics Engineers (2000) *High Level Architecture Interface Specification.* (IEEE 1516.1-2000)

[49] Institute of Electrical and Electronics Engineers (2000) *High Level Architecture Object Model Template.* (IEEE 1516.2-2000)

[50] Institute of Electrical and Electronics Engineers (2000) *High Level Architecture Rules.* (IEEE 1516.3-2000)

[51] Kamin, S. (2003) Routine Run-time Code Generation. *ACM SIGPLAN Notices* 38 (12). 44-56

[52] Kapolka, A. (2003) *The Extensible Run-Time Infrastructure (XRTI): An Emerging Middleware Platform for Interoperable Networked Virtual Environments* (Masters Thesis, The Moves Naval Postgraduate Institution, 2003)

[53] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. (2001) *An Overview of AspectJ* In the Proceedings of the 15th European Conference on Object-Oriented Programming, p.327-353

[54] Kochan, S. (2003) *Programming in Objective-C.* SAMS Publishing: Indianapolis.

[55] Kontio, J., Caldiera, G. & Basili, V. (1996) *Defining Factors, Goals and Criteria for Reusable Component Evaluation* In Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research.

[56] Ladded, R (2003) *AspectJ in Action.* Manning Publications: Greenwich

[57] Laddad. R. (2002) *I want my AOP!* Last Retrieved on January 5, 2009 from: http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html?page=5

[58] Law, A. & Kelton, D. (2000) *Simulation Modeling and Analysis*, 3rd Ed., New York: McGraw-Hill.

[59] Javier Lerch, F., Mantei, M. & Olson, J. (1989) *Skilled Financial Planning: The Cost Of Translating Ideas Into Action*. In Proceedings of the SIGCHI conference on Human factors in computing systems. 121-126

[60] Lorimer, R. (2005) *A Quick Journey Through Spring AOP*. Last Retrieved on January 5, 2009 from: http://www.javalobby.org/java/forums/t44746.html

[61] Lu, T., Lee, C., Hsia, W. & Lin, M. (2000) Supporting Large-Scale Distributed Simulation Using HLA. *ACM Transactions on Modeling and Computer Simulation*, 10 (3). 268-294

[62] Lutz, R. (1997) *A Comparison Of HLA Object Modeling Principles With Traditional Object-Oriented Modeling Concepts* In Proceedings of Fall Simulation Interoperability Workshop. Workshop paper: 97F-SIW-025.

[63] Macannuco, D., D. Coffin, B. Dufault, & W. Civinskas (1999) *Experiences with a FOM Agile Federate* In Proceedings of Spring Simulation Interoperability Workshop. Workshop Paper: 99S-SIW-052

[64] Madina, D. & Standish, R. (2001) *A system for reflection in C++*. Technical Paper. Last Retrieved on March 17, 2004 from: http://parallel.hpc.unsw.edu.au/rks/docs/classdesc/

[65] Martin, R. (2003) *UML for Java Programmers*. New Jersey: Prentice Hall

[66] McCarty, B. & Cassady-Dorion, L. (1998) *Java Distributed Objects*. SAMS Publishing: Indianapolis.

[67] Mellon, L. & West, D. (1995) *Architectural optimizations to advanced distributed simulation* In Proceedings of 27th Winter Simulation Conference. 634-641.

[68] Mellor, S., Balcer, M. (2002) *Executable UML: A Foundation For Model-Driven Architecture*. Addison-Wesley:Boston

[69] Menzler, H., Krosta, U. & Pixius, K. (2000) *HLA in a Nutshell: Ψ-SA Proposed Standard Interface for Simulation Applications* In Proceedings of Spring Simulation Interoperability Workshop. Workshop paper: 00S-SIW-026.

[70] Milosavljevic, B., Vidakovic, M. & Konjovic, Z. (2002) *Automatic Code Generation for Database-Oriented Web Applications* In Proceedings of the second workshop on Intermediate representation engineering for virtual machines. 59-64

[71] Möller, B., Löfstrand, B. & Karlsson, M. (2007) *An Overview of the HLA Evolved Modular FOMs* In Proceedings of Spring Simulation Interoperability Workshop. Workshop paper: 07S-SIW-108.

[72] Morse, K. (2000) *An Adaptive, Distributed Algorithm for Interest Management.* PhD Thesis. University of California, Irvine.

[73] Mowbray, T. & Zahavi, R. (1995) *The essential CORBA: systems integration using distributed objects*. Wiley Publishing: New York.

[74] Newcomb, M. *oHLA Project Page*. Last Retrieved on January 5, 2009 from: http://sourceforge.net/projects/ohla

[75] Object Management Group, *The Model Driven Architecture* Last Retrieved January 5, 2009 from: http://www.omg.org/mda/

[76] Object Management Group, (2001) *UML Specification Version 1.1*. Last Retrieved on January 5, 2009 from: http://www.omg.org/cgi-bin/doc?ad/97-08-11

[77] Object Management Group (2001) *Model Driven Architecture, A Technical Perspective*. Technical Paper. Last Retrieved on January 5, 2009 from: www.omg.org/cgi-bin/doc?ormsc/2001-07-01

[78] Object Management Group (2002) *Unified Modelling Language Specification (Action Semantics)*. Technical Paper. Last Retrieved on January 5, 2009 from: www.omg.org/cgi-bin/doc?ptc/02-01-09

[79] Object Management Group (2005) *What is UML?* Last Retrieved on January 5, 2009 from: http://www.omg.org/gettingstarted/what_is_uml.htm

[80] Panko, R. (1998) *What We Know About Spreadsheet Errors.* Journal of End User Computing, 10 (2), 15-21.

[81] Panko, R. & Halverson, R. (1996) *Spreadsheets on trial: A survey of research on System Sciences* In Proceedings of 29th Hawaii International Conference on System Sciences. 326-332

[82] Parr, S., Radeski, A. & Whitney, R. (2002) *The Application of Tools Support in HLA* In Proceedings of Simulation Technology and Training Conference. 51-55

[83] Parr, S. & Radeski, A. (2002) *Towards a Simulation Component Model for HLA* In Proceedings of Fall Simulation Interoperability Workshop. Workshop Paper: 02F-SIW-079

[84] Parr, S., Radeski, A. & Wharington, J. (2002) *Component-Based Development Extensions to HLA* In Proceedings of Spring Simulation Interoperability Workshop. Workshop paper: 02S-SIW-046

[85] Parr, S. & Keith-Magee, R. (2003) *Making the Case for MDA* In Proceedings of Fall Simulation Interoperability Workshop. Workshop paper: 03F-SIW-026

[86] Parr, S. & Keith-Magee, R. (2003) *The Next Step – Applying the Model Driven Architecture to HLA* In Proceedings of Spring Simulation Interoperability Workshop. Workshop paper: 03S-SIW-123

[87] Perrone, P., Venkata, S. & Chaganti, R. (2000) *Building Java Enterprise Systems with J2EE.* Indianaplois: SAMS Publishing

[88] Pokorny, T. (2004) *fedWS: Web Services Access to Active HLA Simulations* In Proceedings of Simulation Technology and Training Conference. 40-44.

[89] Pokorny, T. & Fraser, M. (2004) *Extending Distributed Simulation: Web Services Access to HLA Federations* In Proceedings of European Simulation Interoperability Workship. Workshop paper: 04E-SIW-034.

[90] Pokorny, T. (2005) *The Model Driven Architecture: No Easy Answers* In Proceedings of Simulation Technology and Training Conference. 185-192.

[91] Pokorny, T. & Cramp, A. (2006) *Web Services Performance Analysis for the HLA* In Proceedings of Simulation Technology and Training Conference. 207-312.

[92] Pokorny, T., Stratton, D. & Smith, P. (2006) *AOP and the HLA: Simplified Federation Development* In Proceedings of Fall Simulation Interoperability Workshop. Workshop paper: 06F-SIW-034

[93] Pokorny, T. (2006) *Open Source and the HLA: I Swear it's Here Somewhere* In Proceedings of Fall Simulation Interoperability Workshop. Workshop paper: 06F-SIW-035

[94] Power, D.J., *DSS Resources History of Spreadsheets*. Last Retrieved January 5, 2009 from http://www.dssresources.com/history/sshistory.html

[95] Rajalingham, K., Chadwick, R. & Knight., R. (2001) *Classification of Spreadsheet Errors* In Proceedings of Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG), 2001.

[96] Roiser, S. & Mato, P. (2004) *The SEAL C++ Reflection System* In Proceedings of Computing in High Energy and Nuclear Physics, 222-225.

[97] Roman, E. (1999) *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*. New York: Wiley Publishing

[98] Ronen, B., Palley, A. & Henry, C. (1989) Spreadsheet Analysis and Design. *Communications of the ACM* 32 (1), 84-93

[99] Rothermel, G., Burnett, M., Lixin, L., Dupuis, C. & Sheretov, A. (2001) A Methodology for Testing Spreadsheet. *ACM Transactions on Software Engineering and Methodology* 10 (1), 110-147

[100] Ruh, W., Herron, T. & Klinker, P (1998) *IIOP complete: understanding CORBA and middleware interoperability*. Addison Wesley: Essex.

[101] Salt, J. (1993) *Simulation should be easy and fun* In Proceedings of the 25th conference on Winter simulation. 1-5

[102] Saleh, M. & Gomaa, H. (2005) *Separation of concerns in software product line engineering* In Proceedings of Workshop on Modeling and Analysis of Concerns in Software. 1-5.

[103] Schriber, T. (1991) *An Introduction to Simulation Using GPSS/H.* New York: John Wiley.

[104] Seila, A. (2003) *Spreadsheet Simulation.* In proceedings of Winter Simulation Conference. 25-30

[105] Siegel, J. (2001) *Developing in OMG's Model-Driven Architecture.* Last Retrieved on January 5, 2009 from: http://www.omg.org/mda/papers.htm

[106] Simulation Interoperability and Standards Organization (2004) *Dynamic Link Compatible HLA API Standard for the HLA Interface Specification Version 1.3 (Java).* (SISO-STD-004-2004)

[107] Simulation Interoperability and Standards Organization (2004) *Dynamic Link Compatible HLA API Standard for the HLA Interface Specification Version 1.3 (C++).* (SISO-STD-004.1-2004)

[108] Shanks, G. (1997) The RPR-FOM. *A Reference Federation Object Model to Promote Simulation Interoperability* In Proceedings of Spring Simulation Interoperability Workshop. Workshop paper: 97S-SIW-135.

[109] Snively, K. & Grim, P. (2006) *ProtoCore: A Transport Independent Solution for Simulation Interoperability* In Proceedings of Fall Simulation Interoperability Workshop. Workshop paper: 06F-SIW-093.

[110] Soley, R. (2001) *MDA Announcement and Technical Briefing.* Recording of Presentation. Last Retrieved on March 17, 2004 from: http://www.omg.org/mda/mda_audio/Soley01.mp3

[111] Steed, M. (1992) Stella, a simulation construction kit: cognitive process and educational implications. *Journal of Computers in Science and Mathematics Teaching*, 11 (1), 39-52.

[112] Straßburger, S. (2001) *Distributed Simulation Based on the High Level Architecture in Civilian Application Domains.* PhD Thesis. University of Magdeburg

[113] Stratton, D., Miller, J. & Parr, S. (2004) *Developing an Open-Source RTI Community* In Proceedings of Spring Simulation Interoperability Workshop. Workshop paper: 04S-SIW-011.

[114] Stytz, M. & Banks, S. (2001) *Enhancing the Design and Documentation of High Level Architecture Simulations Using the Unified Modeling Language* In Proceedings of Spring Simulation Interoperability Workshop. Workshop Paper: 01S-SIW-006

[115] Sun Microsystems Incorporated (1999) *The Java hotspot performance engine* architecture: A white paper about Sun's second generation
performance technology. Technical report. Last Retrieved on January 5, 2009 from: http://java.sun.com/products/hotspot/whitepaper.html

[116] Tolk, A. (2002) *Avoiding another Green Elephant – A Proposal for the Next Generation HLA based on the Model Driven Architecture* In Proceedings of Fall Simulation Interoperability Workshop. Workshop Paper 02F-SIW-004

[117] Tolk, A. (2003) More Stories on the Green Elephant – A Short Introduction to the Model Driven Architecture and its Usefulness for M&S. Simulation Technology Magazine 6 (1) Last Retrieved on January 5, 2009 from: http://www.sisostds.org/webletter/siso/iss_91/art_502.htm

[118] Tudor, G & Zalcman, L. (2006) *HLA Interoperability - An Update* In Proceedings of Simulation Technology and Training Conference. 345-350.

[119] University of Ballarat (2002) *The Air Transport Operations Federation*. HLA Training Course Materials.

[120] Vissim. Product Website. Last Retrieved on January 2, 2008 from: http://www.vissim.com/

[121] Vollmann, D. (2005) *Aspects of Reflection in C++*. Technical Report. (ISO/IEC JTC1/SC22) Last Retrieved on January 2, 2008 from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1751.html

[122] Walls, C. & Breidenbach, R. (2005) *Spring in Action*. Manning Publications: Greenwich.

[123] Weatherly, R., Wilson, A. & Griffin, S. (1993) *ALSP-theory, experience, and future directions* In Proceedings of 25th Winter Simulation Conference. 1068-1072.

[124] Wilbert, D. (1999) *A Tool for Configuring FOM Agility* In Proceedings of Fall Simulation Interoperability Workshop. Workshop paper: 09F-SIW-116.

[125] Wiedemann, T. (2000) *VisualSLX: an open user shell for high-performance modeling and simulation* In Proceedings of Proceedings of the 32nd conference on Winter simulation. 1865-1871.

[126] Wilcox, E. M., Atwodd, J. W., Burnett, M. M., Cadiz, J. J., & Cook, C. R. (1997) *Does continuous visual feedback aid debugging in direct-manipulation programming systems?* In Proceedings of the ACM Conference on Human Factors in Computing Systems. 258-265.

[127] Aspect Oriented Programming Implementations List. Last Retrieved on January 5, 2009 from:
http://en.wikipedia.org/wiki/Aspect-oriented_programming#Implementations

[128] Wittman, R. & Harrison, C. (2001) *OneSAF: A Product Line Approach to Simulation Development* In Proceedings of European Simulation Interoperability Workshop. Workshop paper: 01E-SIW-061.

[129] Yilmaz, L. (2004) On the Need for Contextualized Introspective Models to Improve Reuse and Composability of Defence Simulations. *Journal of Defense Modeling and Simulation: Applications, Methodology and Technology*, 3 (1), 135-145.