On the Definition of Non-Player Character Behaviour for Real-Time Simulated Virtual Environments

Eike Falk Anderson

National Centre for Computer Animation

A thesis submitted in partial fulfilment of the requirements of
Bournemouth University for the degree of

Doctor of Philosophy

Submitted: April 2008

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and due acknowledgement must always be made of the use of any material contained in, or derived from, this thesis.

On the Definition of Non-Player Character Behaviour for Real-Time Simulated Virtual Environments

Eike Falk Anderson

Abstract

Computer games with complex virtual worlds, which are populated by artificial characters and creatures, are the most visible application of artificial intelligence techniques. In recent years game development has been fuelled by dramatic advances in computer graphics hardware which have led to a rise in the quality of real-time computer graphics and increased realism in computer games. As a result of these developments video games are gaining acceptance and cultural significance as a form of art and popular culture.

An important factor for the attainment of realism in games is the artificially intelligent behaviour displayed by the virtual entities that populate the games' virtual worlds. It is our firm belief that to further improve the behaviour of virtual entities, game AI development will have to mirror the advances achieved in game graphics. A major contributing factor for these advancements has been the advent of programmable shaders for real-time graphics, which in turn has been significantly simplified by the introduction of higher level programming languages for the creation of shaders. This has demonstrated that a good system can be vastly improved by the addition of a programming language.

This thesis presents a similar (syntactic) approach to the definition of the behaviour of virtual entities in computer games. We introduce the term behaviour definition language (BDL), describing a programming language for the definition of game entity behaviour. We specify the requirements for this type of

programming language, which are applied to the development and implementation of several behaviour definition languages, culminating in the design of a new game-genre independent behaviour definition (scripting) language. This extension programming language includes several game AI techniques within a single unified system, allowing the use of different methods of behaviour definition.

A subset of the language (itself a BDL) was implemented as a proof of concept of this design, providing a framework for the syntactic definition of the behaviour of virtual entities in computer games.

This thesis is dedicated to Monika Anderson.

Author's Note

I write this at the end of a long journey. For lack of a better word to describe it, the work behind this thesis, as well as the thesis itself has been a journey of discovery, not invention; and while the map is now a lot less empty than it was before (back in 2001), there are still areas of "Terra Incognita" where there be dragons.

Computer games have fascinated me ever since I first came into contact with computers. My interest in computer programming then grew out of a desire to understand the inner workings of games, so that I could modify and create games by myself. Learning to program taught me that programming languages are a powerful tool through which a computer can be made to do almost anything. The CGAL animation system that I was introduced to during my undergraduate studies proved this, as it provided the means to create moving images through the power of programming. Graphics alone, however, do not make a game – there is also the artificially intelligent behaviour of NPCs (Non-Player Characters) within the games.

The majority of NPCs that we (game programmers) send into battle are doomed to suffer a dreadful fate. It is quite obvious that the main cause of an NPC's death is the often bloody confrontation with a human player's avatar. The moral implications of this virtual carnage aside, the question we must ask is whether we have done the best we can in preparing these artificial warriors for battle. The truth is, I do not know. I hope though, that the work described in this thesis will provide a glimmer of hope to NPCs everywhere – not that it will really improve their chances of survival... I am a bad loser and I would hate to see them win. I am content as long as they lose convincingly.

Acknowledgements

This thesis owes its existence to many people who encouraged and supported my research.

First and foremost I need to express my gratitude towards my supervisor, Peter Comninos, for his help, encouragement and guidance. Without his support and advice this project would not have reached its current state. The constant feedback and ideas that he provided me with during the preparation of this thesis were invaluable.

Also, I thank Anargyros Sarafopoulos, my second supervisor, for inspiration, support and the permission to use his GP system for the development of the GP Asteroids scripting system. His expertise and suggestions contributed significantly to the development of this system.

I am also grateful for the comments and suggestions offered by my thesis examiners, Meurig Beynon and Alexander Pasko, which were very useful for the preparation of the final version of this thesis.

I also should mention the ZFX forum admins and moderators (current and former) who provided a valuable source of information and inspiration. First among them the founder of ZFX, Stefan Zerbst, for "mentioning" that a scripting extension to his game project "Pandoras Legacy" might be a good idea, prompting me to develop the ZBL/0 behaviour definition (scripting) language that turned out to be the perfect test bed for a number of hypotheses relating to behaviour definition languages. Regarding the ZBL/0 language I also need to thank the members of the ZFX team for program testing and for the feedback they provided me with, especially Milo Spirig, Sebastian Pech and Oliver Düvel. Their suggestions and comments were always encouraging and very useful for the design of the ZBL/0

system.

Additional thanks go to my fellow PhD students Marina Militadou, Olusola Aina, and more recently Leigh McLoughlin and Steffen Engel, who made life in the research lab bearable and interesting. Sola introduced me to LaTeX and "PhD Comics"[1], both of which have been incredibly helpful – the first turned out to be a useful tool for the preparation of conference papers, as well as this thesis, whereas the second provided some needed comic relief.

To Leigh McLoughlin and Steffen Engel I owe many thanks. They have been good friends, as well as my partners in crime, first on the C-Sheep project – the fluffy stuff of nightmares – and lately on the project that we have tentatively dubbed "Project Flatline".

Finally I would like to thank my family who have supported me throughout this journey. This is especially true for my late grandmother, without whom I would not have got to where I am now.

[1] http://www.phdcomics.com

Contents

CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

Computer games have come a long way since the days of Spacewar[1] [Fleming 2007]. In recent years interactive video games have greatly gained in prominence, and with video games gaining acceptance and cultural significance as a form of art and popular culture, games are now more visible than ever.

Modern computer games aim to immerse the player in a virtual game world by placing him in an interesting and challenging setting that he can interact with, which clearly distinguishes games from other entertainment media. They allow the player to become the narrator and sometimes even the protagonist – either as his virtual self or by assuming the identity of an established character – and tell his own story.

This growth in the popularity of games has been driven by significant advances in game technology, and as a consequence virtual game worlds have become increasingly realistic over the years. Modern games usually employ 3D animated graphics (and 3D sound effects) to provide players with the illusion of realism. A major contributing factor to this end has been a steep rise in the quality of real-time computer graphics, fuelled by dramatic advances in computer graphics hardware.

Whereas once the limitations of the available hardware required ad-hoc solutions, i.e. the development of a new, tailor-made renderer for almost every game,

[1]Spacewar is the first computer game that can be considered the ancestor of modern video games. Created in 1962 at MIT using a DAC PDP-1 computer, it featured two player-controlled spaceships in a deadly duel.

now standardised APIs and functionality for high-end graphics have made the creation of multiple-title, reusable game engines possible.

The stage that graphics have now arrived at leaves little room for significant developments in this field that could lead to an overall improvement of computer games. Furthermore, it is easily recognisable that graphical realism alone does not necessarily make the experience of playing a game realistic. As a direct consequence of this the games industry needs to find other avenues to further improve quality and to distinguish their games. Graphics aside, another very important factor for the attainment of realism in computer games is the behaviour of the characters and creatures that populate the virtual game environments.

This becomes blatantly obvious if the behaviour of computer controlled Non-Player Characters (NPCs) [Olsen 1991] does not "feel right", effectively destroying the illusion of realism.

NPCs are virtual entities inhabiting the game world, whose perception and actions within the game are controlled by a computer program. The behaviour displayed by the NPCs is usually generated with the aid of "artificial intelligence" (AI) algorithms and techniques. The improvement of game AI therefore provides an avenue to achieving the goal of an overall improvement of computer games that is certain to become increasingly important.

There is no single, common method for the implementation of a game character AI. The life-like behaviour of the NPCs that populate the virtual game worlds often requires the combined use of several techniques determined by the desired effect. This kind of artificially intelligent entity is commonly referred to as an autonomous agent[2].

Despite the importance of a good NPC AI in games, over the past decade there have been few changes to the techniques employed by the game developers. While there exist a multitude of possibilities for creating a game character AI, only a relatively small subset of tried and tested methods are used, usually to create a project-specific AI solution for that game's virtual entities.

[2]An agent is a program that has the ability to perceive and to (re-)act. An autonomous agent is a program that has the ability to control itself. Its actions are derived from an analysis of the agent's situation and environment based on its knowledge and experience.

Until recently, the artificially intelligent behaviour for NPCs was almost always hard-coded into the game itself, i.e. the source code for the AI forms an intrinsic part of the game program, and only works for the particular game it was created for. The behaviour of NPCs in games is therefore not easily reusable for other game productions, and generally impossible to transfer to other game genres.

The shift towards data-driven architectures has partially addressed this issue, and the introduction of AI middle-ware now allows a certain degree of reusability, however, the use of ad-hoc solutions for each individual game is still prevalent among game developers.

A comparison between our initial observations on the development of graphics in games, especially the move from individual approaches to more standardised methods, and recent developments in NPC AI development allow us to draw certain parallels:

The introduction of programmable GPUs (Graphical Processing Units) and therefore the advent of programmable shaders for real-time graphical applications in recent years [Lindholm et al. 2001] has shown that with relatively little effort, great advances in the graphical quality of computer games can be achieved. Furthermore, the successive introduction of higher level programming languages for the creation of these shaders [Mark et al. 2003] has demonstrated that even better graphical quality for games is attainable by providing more powerful tools to the developers.

It is our firm belief that to achieve further improvements in the quality of computer games a similar approach will have to be taken for the creation of the artificially intelligent characters that populate the virtual worlds of computer games, i.e. the creation of a high-level programmable system for defining NPC behaviours is the logical next step.

1.1 Aims

There are a lot of different AI techniques that are suitable for computer games and our motivation is not the exploration of new AI techniques. In light of the game industry's trend to embrace data-driven design, however, one of the main

challenges is to efficiently define the behaviour of artificially intelligent characters by placing these definitions in external game assets that are not hard-coded into the game program itself.

In that respect, one of the main objectives of our research was the design of an extendable and preferably modular system which will simplify the interface that allows the creation of virtual entities in computer games that are able to interact with each other and the virtual environment that they inhabit, effectively tying together the available AI techniques.

This interface should take the form of a behaviour definition language, providing a syntax-driven approach to the definition of AI behaviours for the virtual entities in computer games. A program written in this behaviour definition language would therefore become an external asset for the data-driven architecture of the game in which it is used.

This would provide the first step towards the development of a unified software package for creating life-like NPCs in computer games, just as there are software packages for the creation of other game assets like, for instance, three-dimensional animated artwork for games.

1.2 Contribution

The focus and main contribution of this thesis is the design and implementation of a behaviour definition language for virtual entities, suitable for application to NPCs in computer games.

In particular, this work covers the following aspects:

- An investigation of flexible architectures and different interface implementations that enable the exposure of behaviour definition capability to computer game engines, making the creation of reusable behaviours for virtual entities possible.
- The development and implementation of several behaviour definition languages for virtual entities, evaluating different approaches and implementations.

- Dependent on the results of the above, the design and prototype implementation of a game-genre independent behaviour definition language, exposing different methods of behaviour definition, including the definition of virtual entities as well as elements of their environment that they can interact with, through a unified software interface that will allow existing software to be extended to use this system for behaviour definition.

1.3 Thesis overview

This thesis is divided in three parts.

Part 1, starting with the chapter following this introduction, examines the application of artificial intelligence techniques and scripting systems in computer games, showing how those different subject areas are directly related to our work. In particular, **chapter 2** focuses on AI in general, and especially AI in computer games, offering an insight into the use of artificially intelligent entities in computer games and further elaborating some of the points made in this introduction. It reviews common techniques, details problems faced by game AI and considers possible solutions. The discussion pays particular attention to classical AI techniques that are permeating into computer game AI and highlights the most promising game AI methods.

Chapter 3 discusses data-driven architectures for computer games, focussing on the manifestation of the data-driven design philosophy in the use of scripting languages.

Part 2 explores the general requirements for the design of behaviour definition languages for use in computer games as well as existing approaches to behaviour definition using syntactic methods.

Chapter 4 reviews common approaches to the implementation of NPCs in computer games, noting how these game AI techniques are usually applied to satisfy the demands placed on the AI by modern computer games.

Chapter 5 examines requirements and design principles for the creation of behaviour definition languages. It also explains the considerations and ideas that

have directly influenced our work, including educational mini-languages and dedicated AI (scripting) languages that fit into the category of behaviour definition languages.
In light of these, **chapter 6** provides an overview of scripting systems and scripting languages with a specific focus on existing solutions using generic embeddable scripting languages for use in computer games.
Following this. some of the behaviour definition languages that have been created in the course of our work are discussed in **chapter 7** , which includes the ZBL/0 programming language that we developed for inclusion in a book on game development [Zerbst et al. 2003].

Part 3 charts the design and implementation of the behaviour definition system which lies at the core of our solution, the AvDL language and its SEAL subset.
Chapter 8 provides a brief overview of AvDL, the Avatar Description Language, while the topic of **chapter 9** is the SEAL subset of AvDL which enables the system to make use of the most promising game AI techniques introduced in chapter 2.
Chapter 10 describes the design of the SEAL/AvDL Virtual Machine which executes SEAL/AvDL programs, as well as the implementation of the interface to the virtual machine that allows it to be embedded within a host application.
Finally, **chapter 11** provides a discussion of our system, integrating it with the findings of part 2, followed by the presentation of conclusions on this thesis in **chapter 12**.

The main body of the thesis is followed with several appendices that contain additional information on the syntax and usage of the behaviour definition languages that were created as part of this research project.

Part I

Game AI

Chapter 2

Intelligent Non-Player Characters

One of the earliest developments since the appearance of computer games has been the introduction of AI to provide human players with a challenging, involving and – most importantly – with a "fun" experience. The first games with computer controlled players started using AI related techniques for the creation of believable adversaries or enemies to compete with or fight against the human player if no real human opponent was available to take its place. Depending on whether these AI players were tactical opponents in classical board-games or monsters in role-playing games or arcade games, the methods used for creating the AI were different, but their purpose was ultimately the same – to create intelligent NPCs that are life-like opponents for the human player.

2.1 Artificial Intelligence in Computer Games

When we refer to AI in computer games, that which we refer to is not truly AI – at least not in the traditional sense of the term. The techniques applied to computer games are usually a mixture of AI related methods whose main concern is the creation of a believable illusion of intelligence.

2.1.1 Artificial Intelligence

AI is one of the oldest branches of computer science, almost as old as computer science itself, although it took some time for the field to be recognized as such. Research in artificial intelligence even existed a very long time before the term "artificial intelligence" was first used, with roots going as far back as ancient Greece when philosophers (Socrates, Plato, Aristotle) discussed the way in which the human mind functions and how intelligent decisions are made [Anderson 2003a]. The study of what we now call AI is very much rooted in the study of philosophy and the quest for the understanding of the human mind and body. The term "artificial intelligence" for this field of research was coined in 1956 when a number of researchers interested in the study of intelligence and neural networks took part in a workshop (Dartmouth Conference) organised by John McCarthy [1955]. Since this early research there have been numerous attempts towards the creation of AI, often depending on whatever definition of the term AI was used. Each distinct interpretation of the term "artificial intelligence" is associated with different approaches to creating AI. In turn. each of those approaches is more or less suitable for the different areas of AI research. Independent of the definition of AI used, however, the problem they all try to solve and their ultimate goal is the understanding and creation of intelligent programs. The dictionary definition for "artificial intelligence" is "the study of the modelling of human mental functions by computer programs" [Collins 2001a]. A closer look at this branch of computer science, however, shows that this description is far less than accurate. AI is not necessarily confined to the simulation of methods that are biologically accurate or biologically possible [McCarthy 2007]. A different definition for AI for instance is the ability "to solve problems that would require intelligence if solved by humans" [Johnson and Wiles 2001], or the ability of a system to adapt to its environment through learning.

There are many who question if AI can ever reach a level of intelligence that could be compared to that of a human, and while not everyone thinks of human-level intelligence as a goal for the development of AI, human-level AI is especially interesting for games as it promises a human-like opponent for the human player. An early measurement for the presence of a kind of human-like intelligence that

would comply with these aims is the Turing test[1] [Turing 1950]. If a program manages to pass the Turing test, i.e. manages to convince a human that it is human (and therefore intelligent) itself, that program should be considered somewhat intelligent. John Searle's "Chinese Room argument" [Searle 1980], however, suggests that the Turing test is overrated and alone would not be enough to allow judgement of the artificial intelligence of a computer program. It states that just by following a set of rules regarding a language one does not even understand (Chinese in the case of his argument), one might be able to pass the Turing test in that language which would mean that the Turing test itself could not be used as a measure for intelligence or understanding. A further argument against the Turing test is that during the experiment the interrogator knows that he is participating in a game, resulting in his anticipation and expectations generating some form of bias in which the interrogator's imagination makes him perceive intelligence where there is none.

This classical AI goal, aiming for human-like intelligence, is still far away from reaching a solution despite many advances in technology and half a century of research. The fact that an increasing number of the AI techniques developed towards this goal are "spilling over" into computer game AI might suggest that in the future the ability of NPCs to project the illusion of life-like behaviour will increase substantially. However, it cannot automatically be taken as an indicator for these AI techniques' suitability or success, as long as the question of AI itself remains unanswered.

[1]The Turing test, also known as the imitation game, can be explained in simple terms. It requires a set-up of a closed room containing a human test person (the interrogator) at a computer terminal running a chat program, which has two connections. One connection is to a second human operated terminal in a different room and the second connection is to a computer running an intelligent program which pretends to be a human (chatterbot). The interrogator now has to decide which of the two chat partners is human and which one is the chatterbot. If the chatterbot manages to convince the interrogator that it is human, then it has passed the Turing test.

2.1.2 Smoke and Mirrors (Game AI)

The problem that AI in computer games tries to address is a different one, since here its aim is not the creation of actual intelligence, but rather the illusion of intelligence [Scott 2002b]. The behaviour of NPCs only needs to be believable to convey the presence of intelligence and to immerse the human player in the game world. As this means that very little real reasoning is involved, some might argue, that the term "artificial instincts" might be a better description for the level of intelligence that is found there, mainly due to its reactive nature. In the light of some games, the acronym AS for the term "artificial stupidity" might be even more appropriate.

As a rule of thumb one can say that the creation of a simple AI for a computer game is a relatively easy task, as the human brain is easily fooled. With very little effort, an observer can be convinced of the "intelligent actions" of a fairly basic NPC, as long as these actions appear plausible, in a very similar way to the "uncanny valley" phenomenon encountered in the study of the effect of humanoid robots on human observers [MacDorman 2005; Hayward 2007]. The effect of a complex AI, on the other hand, is actually quite invisible and will hardly be recognised as such, suggesting that the concept of "less is more" can be applied to AI in computer games. Its main requirement for creating the illusion of intelligence is perception management, i.e. the organisation and evaluation of incoming data from the AI entity's environment. This perception management mostly takes the form of acting upon sensor information but also includes communication between or coordination of AI entities in environments which are inhabited by multiple NPCs which may have to act co-operatively.

The problems encountered by an AI entity in a game are a combination of the virtual "real-world problems" that face a human game player, as well as various problems that are specific to the various techniques that were used to build the AI. In many cases game AI is deterministic, using rule-based systems which allow game designers to exert a high level of control over the NPCs' behaviour, but while most game AI solutions are provided by a small number of tried and tested methods, a convergence of techniques from a wide range of different fields can be found in computer game AI. These include but are not limited to:

- Traditional (academic) AI [McCarthy 2007], as described above (see Section 2.1.1).

- "Artificial Life" (AL), the study of "multi-agent systems that attempt to apply some of the universal properties of living systems to AI agents in virtual worlds" [Tozour 2002b], which includes some machine intelligence techniques related to emergent behaviours like flocking [Reynolds 1987] and evolutionary techniques like Genetic Algorithms (GA) or Genetic Programming (GP) [Koza 1992], both of which are automated techniques that produce algorithms by using a process that parallels evolution through natural selection, i.e. a simulation of life.

- Robotics, especially the cognitive robotics techniques that allow a robot to orient itself and navigate in the world.

- Empirical observation of behaviour. Much information on behaviour can be acquired through the study of nature. The science of ethology, the biological study of behaviour, provides valuable insight into the behaviour of animals [Roberts 1971], some of which can directly be applied to the creation of life-like NPCs in computer games.

A game AI is usually comprised of an amalgamation of possible solutions for each of the combined problems from the different fields. The exact combination required for a solution depends on the role assigned to the AI in the game and subsequently the behaviour which a human player might expect from that type of virtual entity.

2.2 The Roles and Requirements of AI in Computer Games

To gain an understanding of what is expected of an artificial character in computer games one needs to look at how over time NPCs have evolved into the AI entities that one can encounter in modern computer games. The artificial entities populating the virtual worlds of computer games will typically take on one of the following roles [Laird and van Lent 2001; Glasser and Soh 2004]:

- The human player's (tactical) enemy (unit or individual). This is the original AI role in computer games. While the most challenging opponent for a human player is another human being, human opponents are not always available, which was especially true before the proliferation of personal computer networks and networked multi-player games, requiring the use of good AI enemies instead. Starting with the 'intelligent' monster in the game "Hunt the Wumpus" [Yob 1975] to the enemy NPCs in modern first person shooter (FPS[2]) games, AI controlled entities have been used as the core method for providing the challenge for the human player.

- The human player's partner (team-mate). This kind of AI entity is closely linked to the rise of the team-based networked multi-player game. In the early 1990s the development of the internet and improvements and cost reductions in networking technology which led to the widespread introduction of local area networks (LANs) made the creation of games in which multiple players could engage over a network connection possible [Falise 2000]. While in the first of these multi-player games all of the players were opponents, it did not take long, however, for different ways of playing than just fighting against each other over a network to emerge. The co-operation of some players and the subsequent team formation (referred to as clans) have led to games in which large teams engage each other competitively. The overwhelming success of the team-based multi-player games that were created in reaction to this development prompted game developers to attempt to generate the same kind of sensation and experience in single-player games. Artificial team-mates that act in league with the player (collaborative NPCs) have evolved as a direct result of this trend [Kushner 2002].

- The supporting character (incidental), a character that enriches the virtual game world without actively having to contribute to the plot of the game.

[2]An FPS or First Person Shooter game is an action video game in which the player experiences the gameplay from the viewpoint of the protagonist. This type of game usually involves the exploration of some sort of building complex and frequent skirmishes with other players or NPCs. Falise [2000] presents a study of the FPS game genre, providing an overview of its history.

The resources that have become available to games as computers have become more powerful have been the addition of background characters and creatures. Just as in a film, "extras" such as flocks of birds in the virtual sky above or people going about their business in the background of the action, generate a sense of reality which deepens the player's immersion within the game world of games that are continuously growing more complex. In the literature this kind of neutral synthetic entity is sometimes referred to as a Non-Player Character (NPC) [Siem 2006], in that case meaning a character that does not act like a player (human or computer controlled). We prefer the meaning of NPC to include any kind of virtual entity that is not human-player controlled [Yue and de Byl 2006], making the support character a kind of ambient NPC [Cutimitsu et al. 2006].

- The strategic opponent, an artificial entity often encountered as the human player's adversary in real-time strategy (RTS[3]) games [Scott 2002a]. Different from other intelligent characters, this kind of NPC does not usually have a single avatar within the game world but instead is represented by a variety of smaller units under its control. Its tasks within the game include research and resource management, unit construction and training, as well as combat control. The responsibility for carrying out these tasks is normally divided among a number of interrelated AI subsystems which are under overall control of the strategic AI player. The strategic opponent NPC is therefore one of the most complex AI entities found in modern computer games. Path planning and decisions making, comprising of terrain analysis and strategic reasoning, are carried out on a much higher level than found in normal NPCs. A number of RTS games therefore share a number of features with real-life military simulations [Atkin et al. 1999]. However, while at first sight the RTS AI does seem to be very different from the FPS game NPC, many of its underlying concepts are the same.

[3]An RTS or Real-Time Strategy game is a strategy game which is not played round-based but in real-time, i.e. all of a player's units and his opponents have to be directed/make choices on the fly, while all action takes place simultaneously.

- The observer (commentator, tutor or director), an often omniscient entity that provides narrative commentary of the human player's actions and in some cases attempts to guide the human player or NPCs towards the completion of his tasks within the game world [Forbus and Hinrichs 2006]. A recent incarnation of this type of entity are the "intelligent" cameras found in some games that aim to focus the human player's view of the game world onto important events [Kharkar 2004].

The actions of an NPC are governed by its "behavioural model". This defines how the game character reacts to any input it receives from its environment. The interpretation of these inputs depends on the way that this information is exposed to the AI entity and its domain knowledge, i.e. the NPC's perception and understanding of the virtual world it occupies. It is common for games to use high-level inputs that carry a lot of implied information, which can result in believably intelligent behaviour even if only a very simple and basic decision making process is used [Welsh and Pisan 2005], provided that the NPC has the required domain knowledge.

2.2.1 The NPC World Interface

Providing this domain knowledge is important and can be problematic. An NPC's AI needs to be able to clearly map – or anchor – the NPC's environment to its understanding of this environment. Coradeschi and Saffiotti [1999] discuss this "anchoring" problem in the context of autonomous robotics in real environments. They especially stress uncertainty as being the main difficulty in matching real-life sensor data to the symbolic representation of knowledge by the AI. Fortunately this problem is a lot less prevalent in the completely self-contained, virtual environment of a computer game world. Through a game's world interface the incoming sensor data can be controlled to a much higher degree than real-life sensor data, considerably simplifying the matching of sensor information to stored knowledge, which in many cases can be directly mapped to one another. The process of providing this knowledge in the first place, however, still remains quite complex and there are different possible solutions:

1. All associations can be explicitly defined. This is the simplest method but also the least feasible if the AI resides in a large and complex environment, as the amount of data that would have to be provided would be too large. This approach only works in very small or simple scenarios.

2. Associations can be generated. For this various techniques can be used. One way to achieve this would be to employ some kind of learning technique like reinforcement learning which has been successfully implemented in commercial computer games [Johnson and Wiles 2001]. Another possibility would be the use of emergent behaviour techniques like evolutionary algorithms.

3. The environment can be annotated (see Section 2.3.4.4). An annotated environment with smart objects holds all the information necessary for the NPC to interact with it. As a result the NPC can be less complex which not only benefits the development process but also makes the NPC's AI "infinitely extensible" [Orkin 2002], making this method for simplifying the creation of intelligent NPC behaviour a promising game AI technique [Rabin 2004].

2.2.2 AI in Real-Time Computer Games

A major difficulty facing the developers of a computer game AI is the requirement for the NPCs to work in real-time, i.e. concurrently with the human player's interaction with the virtual world and without the dedicated "thinking" cycle for decision making which is available to AI entities in round-based games. The AI has to be made to work so that to the human player it looks like the NPCs are making decisions as they play along. Resource restrictions are an important factor as even at the current rate of advances in computing power, there are still limits to memory and processor (CPU – central processing unit) capabilities and this automatically excludes a number of AI techniques from being used in games, as it would be unacceptable for an NPC to spend minutes of game-time with decision making.

Another problem, which is closely related to this real-time requirement for game AI, is the fact that the AI has to share the computer's processing resources with the rest of the game which will include graphics, input processing, sound processing and synchronisation issues arising from networking. In early computer games, AI was given very little importance and was therefore allocated only little processor time. Only after the development of graphics accelerators in the mid-1990s, when more and more elements of the graphics pipeline were redirected onto dedicated graphics hardware, AI acquired a higher priority and with it additional resources. At first CPU budgets for AI exploded and a number of games spent up to 30% of their processor time doing AI calculations, but this has now levelled off at about 10% of CPU time [Woodcock 2001].

The exact range of problems that an NPC within a computer game has to solve depends on the context in which it exists and the virtual environment in which the game takes place. The tasks which need to be solved in most modern computer games and to which the intelligent actions of NPCs are usually restricted to (by convention rather than technology) are [Anderson 2003a]:

- decision making
- path finding (planning)
- steering (motion control)

2.2.2.1 Human-like NPC Intelligence

Until recently the unique selling point for many video games used to be the quality of graphics and the number of polygons that could be displayed simultaneously on screen. The realisation that graphical realism alone does not make a good computer game has replaced this development trend with a drive to improve the complexity and therefore the believability of the artificial characters that populate the virtual game worlds. NPC behaviour that appears natural adds more life-like qualities to the NPC and makes it seem more realistic. As a crucial factor for the success and popular acceptance of a computer game this has now become more important than ever. Laird and van Lent [2000] argue that the intelligence displayed by NPCs in computer games will ultimately have to reach a human level

at some point in the future, to keep entertaining human players. To achieve this, NPC AI will have to become scalable, i.e. less restrictive and less deterministic than current implementations allow for. Attempting to realise this with current hardware however still results in a number of real-time performance problems. The decision cycle of human-like NPCs can be decomposed into three steps that are constantly executed [van Lent et al. 1999; Wright and Marshall 2000]:

1. sense/perceive (accept information about the environment – sensor information)

2. think (evaluate perceived information & plan appropriate actions)

3. act (execute the planned actions)

Van Lent and Laird [1999] suggest that a system for the creation of this kind of NPC would therefore consist of three components:

1. An inference machine which would constantly execute the NPC decision cycle. This would have an internal memory for remembering goals, which is one of the necessary preconditions for human-like behaviour. Its requirements would be:

   - to use reactive agents
   - to be context specific
   - to be flexible
   - to be realistic
   - to be easy to develop

2. A world interface to the underlying game engine which should mimic the human player's interface as closely as possible, i.e. provide the NPC with all the information (or a representation thereof) that is provided to human players, i.e. audio & visual data, and the controls that allow the NPC to interact with its environment in a similar fashion to the human player.

3. A knowledge base, to provide the NPCs with the necessary domain knowledge, allowing the NPC to correctly interpret its situation in the game world and therefore to make meaningful decisions to inform its actions.

Their rule-based Soar (State, Operator And Result) agent architecture implements such an inference machine. Soar was originally developed as a cognitive architecture for building realistic AI entities with strong military applications. In recent years it has also been used to create NPCs for various FPS games[4]. For example, Soar agents created for the FPS game Quake2 have the ability to anticipate a human player's actions and to adjust their actions accordingly to counter the human player's moves [Laird 2001]. In these games the Soar engine which provides the NPC's run-time environment uses a network connection to communicate with a plug-in[5] to the game engine. This plug-in only provides an interface between the Soar engine which runs remotely with the game engine into which it is plugged in.

2.2.2.2 NPC Complexity vs. NPC Performance

The use of the Soar architecture for computer games is not an ideal one. Soar controlled NPCs are so computationally expensive that it would be very hard for more than one NPC to run on a single computer at the same time. The focus of research into games using this architecture has mainly been on the cognitive capabilities of Soar NPCs by adding learning and some prediction methods to the system to improve the NPCs themselves. While this has certainly made them appear more realistic, it has largely ignored the real-time requirement of computer games, making the Soar architecture unsuitable for general deployment in computer games. Khoo and Zubek [2002] argue that the Soar approach to achieving human-like intelligence for NPCs is over-ambitious and that similar results could be achieved by using a combination of more conventional and inexpensive NPC creation techniques. Observation of FPS game players has shown that the perception of NPC intelligence and skill is determined by reaction (decision) time

[4]http://www.soargames.org

[5]A plug-in is an external software module which is not part of a program but which can interface with the program to provide it with additional functionality. Plug-ins are often implemented as dynamically linked libraries which a program can load during run-time.

and aiming accuracy [Laird and Duchi 2001]. This automatically disqualifies the use of complex – and therefore slow – reasoning algorithms, which is why Khoo and Zubek suggest that a behaviour-based approach from robotics would be more suitable. One result of their work is a successful NPC called "Groo" which was created for the FPS game Half-Life. It interfaces with the Half-Life game engine through a plug-in using the FlexBot [Khoo et al. 2002] plug-in API (application programming interface[6]). The control program for the Groo game-bot itself is written in the GRL programming language [Horswill 2000] (see Chapter 5, Section 5.1.1.2) from robotics which in turn is compiled into native C++ source code for use with FlexBot. A further development of NPCs using this technology is the Half-Life game-bot Ledgewalker [Khoo et al. 2002] which confronts human players with an effective opponent NPC with many qualities which are perceived to be human-like.

2.3 Game AI Techniques – The State of the Industry

Just like computer games have come a long way, so have the AI techniques that are employed within those games, many of which are derived from traditional AI methods. Some of the more proven and successful techniques have changed little over time and those techniques are almost always the first choice of developers when they need to implement AI in their games. However, since the early 1990s an increasing number of novel ideas and methods for game AI have filtered into the game development process [Sweetser 2003]. The greatest changes in the use of AI in games however have involved the selection of AI to solve different problems rather than the choice of AI techniques.

[6]An application programming interface (API) provides the programmer with an interface to a group of related functions that are usually located within a library of functions. The interface in this case is the description of data types, return types and formal parameters to functions and methods (if object orientation is used).

2.3.1 Rule Based Techniques

Rule-based techniques are the oldest and most commonly found AI methods used in computer games. They can be implemented with relatively little effort and they provide a robust and reliable solution to a wide range of problems but are often used for decision making.

2.3.1.1 Finite State Machines

Finite state machines (FSMs) are the most commonly used type of AI used in games [Fu and Houlette 2004]. They arrange the behaviour of the NPC in logical states – defining one state per possible NPC behaviour – of which only one, the NPCs behaviour at that point in time, is active at any one time. A state is a Boolean value which is either active or inactive – 'on' or 'off'. When the current behaviour needs to be changed to a different behaviour, for example a transition from a guarding stance to an attack on the closest opponent, the FSM will switch between the states. It is relatively simple to program a very stable FSM that may not be very sophisticated but that "will get the job done". The main drawback of FSMs is that they can become very complex and hard to maintain, while on the other hand the behaviour resulting from a too simple FSM can easily become predictable. To overcome this problem sometimes hierarchical FSMs are used. These are FSMs where each state can itself be an FSM.

2.3.1.2 Fuzzy State Machines

Fuzzy state machines (FuSMs) are a permutation of FSMs which uses fuzzy logic instead of Boolean logic [McCusky 2000]. As a result states in FuSMs are not limited to existing in one of the two states 'on' or 'off' but they can hold an intermediate value. This means that at any one time more than one state may be active and to some degree be on and off. While this makes the construction of FuSMs slightly more complicated than the creation of an FSM the existence of simultaneously active states greatly reduces the predictability of the resulting behaviour. It also dramatically reduces the complexity of the state machine, as a wider range of different behaviours can be encoded with fewer states. FuSMs are

a relatively new game AI technique that can be used in almost all of the areas in which FSMs are usually found.

2.3.2 Knowledge Based Techniques

Knowledge based techniques are rarely used on their own when it comes to game AI, but they are often used as subsystems of game AI. This would include terrain analysis techniques within strategy games such as influence mapping [Tozour 2001] which allow a strategic AI in a war-game to assess the current situation, to identify choke points for ambushes [Higgins 2002b] or to position its troops on the virtual battlefield. Related to this are the search strategies that are frequently used for path finding for NPCs in a wide range of games.

2.3.2.1 AI Planning

Considered a promising game AI technique [Rabin 2004], planning in games is often performed by using a search algorithm on a knowledge base, representing an NPC's domain knowledge. In computer games, this has mainly been implemented as a method for path finding to facilitate NPC navigation in virtual game worlds, but recent developments aim to apply planning to NPC decision making. While there exist many search methods for path finding, such as Dijkstra's algorithm [Dijkstra 1959], for path planning in games the algorithm of choice is the A* algorithm [Stout 2000] (see Chapter 4, Section 4.3.3) which is optimal, i.e. proven to find the optimal path in a weighted graph if an optimal solution exists [Dechter and Pearl 1985].

More general planners use a notation based on the representation of initial and goal states and the operators or actions required to reach the goals, as is the case with the pioneering STRIPS (STanford Research Institute Problem Solver) program and language which has provided a template for many modern AI planning systems [Russel and Norvig 1995]. Planning can be a complex and time-consuming task that may not be fully computable within the time available in the update-cycle of a real-time computer game, requiring the computation to be "staggered" [Evans 2001], i.e. distributed over several update-cycles to spread the workload of the CPU. This process, known as time-slicing, usually involves

the careful management of AI processes that need "to be dynamically suspended and reactivated" [Wright and Marshall 2000], which can be achieved using multi-tasking techniques usually associated with operating systems.

2.3.2.2 Goal-Oriented Techniques and Goal-Oriented Action Planning

Goal-directed behaviour is one of the simplest forms of nondeterministic behaviour. A goal is the end-state of a set of goal-directed actions. Dybsand describes it as a technique in which an NPC "will execute a series of actions ... that attempt to accomplish a specific objective or goal" [Dybsand 2004]. Goal-oriented techniques have only recently been introduced into computer game development and so far, goal-oriented methods for creating NPC behaviour are employed in only a small, but steadily growing number of commercial games. In its simplest form, goal-orientation can be implemented by determining a goal with an embedded action sequence for an NPC. This action sequence, the NPC's plan, will then be executed by the NPC to satisfy the goal [Orkin 2004a]. Solutions that allow for more diverse NPC behaviour can improve this by selecting an appropriate plan from a pre-computed "plan library" [Evans 2001] instead of using a built-in plan.

More complex solutions use plans that are computed dynamically, i.e. "on the fly", as is the case with Goal-Oriented Action Planning (GOAP) [Orkin 2004a]. In GOAP the sequence of actions that the system needs to perform to reach its end-state or goal is generated in real-time by using a planning heuristic on a set of known values which need to exist within the NPC's domain knowledge. To achieve this in his implementation of GOAP, Orkin [2004b] separates the actions and goals, implicitly integrating preconditions and effects that define the planner's search space, placing the decision making process into the domain of the planner and therefore relieving the designer of the need to micro-manage game logic.

In GOAP the representation of the search space can be augmented by associating costs with actions that can satisfy goals, turning the NPC's knowledge base into a weighted graph, allowing the use of path planning algorithms such as A* that find the shortest path within a graph as the planning algorithm for the NPC's high-level behaviour [Orkin 2006]. This has the additional benefit of

greater code re-use as the planning method for high-level decision making, as well as path planning is the same and can therefore be executed by the same code module [Orkin 2004b] if the representations of the search space are kept identical.

2.3.3 Machine Learning and Emergent Behaviour

Recently the use AI techniques that involve machine learning in games to achieve emergent behaviour has become more frequent and surprisingly effective [Graepel et al. 2004]. The implementation of systems that "learn to play good" can be done without too much effort; however, their unpredictability makes them unsuitable for many games. The danger with learning algorithms is always that instead of making the AI seem smarter by behaving clever, it could in fact learn to behave more stupidly by misinterpreting its inputs. To prevent this from happening the NPCs need to be trained to act in a desirable manner by the game's developers. This learning is usually done before the game itself is published, often using automated off-line calculations, with the commercial product then only using the locked-in, previously learned behaviour, while the learning itself is disabled.

2.3.3.1 Artificial Neural Networks (ANNs)

Neural networks are used to emulate the functionality of human and animal brains. In an artificial neural network the neurons are modelled using interconnected nodes that are able to make new connections, which allows the network to learn and improve itself. Using a neural network can enable games to adapt to the way that the player plays by updating itself during gameplay. As such they have been used in strategy games but they have also been successfully implemented in adventure games or action games, allowing artificial entities to improve their skills in line with the human player's performance.

2.3.3.2 Decision Trees

Decision trees that grow as they learn new information are another machine learning method that is used in computer games. They are one of the most reliable and robust learning methods available and usually the preferred choice if a game AI requires to predict future outcomes or classify situations. When it is generated the

decision tree will store situations and their outcomes within its nodes, allowing it to "remember" the best course of action in case a similar situation is encountered in the future. In games, they have been generated using reinforcement-learning (gathered from the human player's reactions to NPC behaviour).

2.3.3.3 Evolutionary Techniques

Evolutionary techniques are the least often used machine intelligence methods used in computer games. In these techniques a basic initial set of problem solving strategies for NPCs is usually evolved over time using a range of selection methods as well as random mutations, which are then evaluated until an optimal solution is found. While these solutions are usually very robust and reliable it can take a long time for a program to reach the desired level of competence which makes evolutionary techniques unsuitable for most real-time games. Nevertheless a number of games have made use of evolutionary techniques like genetic algorithms (GA) and genetic programming (GP) that have been used for evolving agents for a number of games, including arcade games [Anderson 2002]. GP has so far been applied experimentally to a number of different computer game scenarios. Among these are classic video games like Pac Man [Koza 1994] or Tetris [Siegel and Chaffee 1996]. In these experiments game playing behaviour has been evolved in modified game environments. Most of the game versions used have been round-based, i.e. the computation of actions in the game are performed while the game itself is paused. Gameplay resumes only after those computations have finished, and only lasts until the pre-calculated actions have been executed. This is in contrast to real-time games in which all actions have to be calculated "on the fly". One of the few attempts to apply GP to a real-time game (RoboCup Soccer) is documented by Luke [Luke et al. 1998; Luke 1998]. The methods employed for that experiment bear some similarities to our own experiments [Anderson 2002] (see Chapter 7, Section 7.1).

2.3.4 Extensible NPC Intelligence

A recent trend in computer games is to make them extensible by allowing users to modify them to their needs, one of the main areas for doing so being the definition

of game AI. There are several methods with different levels of complexity that can be used to achieve this.

2.3.4.1 Parameter Tweaking

The simplest way for modifying AI behaviour is by modifying the rules that are used internally by the game AI. This is usually done by setting internal program parameters that determine the behaviour of NPCs to given values. There are a number of games that employ this technique – some games even have graphical user interfaces to make this as simple as possible. Other games employ very simple initialisation scripts (see scripting systems below) to achieve this effect [Tapper 2003].

2.3.4.2 Plug-In Interfaces

As mentioned above (see Section 2.2.2.1), some games contain software interfaces that can be used for writing plug-ins that can change the AI of NPCs in the game [Laird 2001], effectively allowing parts of the games to be reprogrammed. For this purpose, some games even have complex SDKs (software development kits) to simplify the modification of the game behaviour.

2.3.4.3 Scripting Systems

Many new games contain complex scripting systems (see Chapter 6) that allow the game AI to be defined or extended. Through scripting, game modification without the need for the program source code to be recompiled, a task that can be accomplished by a game designer alone, becomes possible. This enables the introduction of "parallel development", which means that the programmers' time is freed up as they no longer need to concern themselves with design elements which designers can now manipulate themselves with scripts [Huebner 1997]. A type of scripting language which is domain specific to the creation of NPC intelligence is the behaviour definition language [Anderson 2004] (see Chapter 5). As their name suggests, behaviour definition languages are used to define the behaviour of virtual characters – often in the form of programs running on

a virtual machine[7] which interfaces with the character controls within the game engine.

2.3.4.4 Annotated Environments

A number of games now use annotated environments ("Smart Terrain") to simplify the simulation of intelligent behaviour. If the environment of the NPC holds all the information necessary for the NPC to interact with it, the NPC can be less complex which allows for the rapid development of game scenarios [Cornwell et al. 2003]. This use of "annotated" objects [Doyle 1999] to make up the virtual game world greatly benefits the development process and also makes the NPC's AI highly extensible. The idea of annotated environments is based on the theory of affordance (or affordance theory) that was developed in the fields of psychology and visual perception. Affordance theory states that the makeup and shape of objects contains suggestions about their usage. A real world example would be a mug whose handle "affords" to be gripped to pick up the mug. Transferred into the context of a computer game, this means that the objects in the virtual world contain all of the information that an NPC will need to be able to use them, effectively making the environment "smart". In the game "The Sims" these "Smart Objects" [Peters et al. 2003] were used for behaviour selection to great effect. This means that most of the AI is not actually programmed into the Sims characters but into their environment. An object will broadcast information about itself to the entities in its proximity, including all instructions that are necessary to enable meaningful interaction between the NPC and the object [Forbus and Wright 2001].

2.3.5 Hybrid Techniques – Agents, Animats and Avatars

The literature often refers to computer game NPCs as agents. Although the term agent is used frequently, there is no single definition for it, but generally speaking an intelligent agent is "anything that can be viewed as perceiving its environment

[7] A virtual machine (VM) is a program that emulates the functionality of a whole computer system. It provides applications with a level of abstraction above the actual hardware (and the operating system) of the computer.

through sensors and acting upon that environment through effectors" [Russel and Norvig 1995]. As the choice of terminology shows, a substantial amount of research using the concept of agents has been carried out in robotics, however in terms of software agents or "softbots" this means a program (module) which is able to collect information about its surroundings and evaluate this data using whatever AI method seems appropriate, resulting in a plan of action which it will then carry out – in effect a decision-making entity. The agents that are referred to most often in the context of computer games are autonomous agents [Nareyek 2000]. Autonomous agents are agents that are self contained, i.e. agents that base their actions upon the information that they are able to gather themselves and their own knowledge. They do not have inputs that allow for external control but they are perceiving, "thinking" and acting by themselves. Using this definition one can clearly see that almost all NPCs in modern computer games can qualify as agents. This can be taken further by transforming the autonomous agents into embodied systems, i.e. virtual beings that interact with their environments using their bodies which take the place of abstract sensors or effectors. These truly autonomous NPCs are called animats [Champandard 2004]. The definition of animats is very close to what might be regarded as the ideal NPC, as it is a believable virtual entity. However, we think that to describe this kind of entity that could be considered the ultimate NPC a different term should be used. The dictionary definition for the word avatar is "the manifestation of a deity ... in human, superhuman, or animal form" [Collins 2001b]. This meaning has been transferred – mainly in multi-player computer games – onto the visual representations of the players within the virtual environments of such games. However, we strongly believe that the meaning of the term avatar in the context of computer games should be expanded to also include virtual characters or virtual creatures which can interact with other avatars and the virtual environment they populate. We come to this belief because in a number of MMOGs (Massive Multi-player Online Games), the human player's avatars will sometimes interact with very human-like NPCs that behave similarly to other players' avatars. The boundaries between player and NPC are effectively blurred and – depending on the realism of the performance of the NPC – it may be hard to distinguish between human player and NPC. This kind of avatar could be called a most human-like

NPC. Therefore in the context of computer games an alternative definition of the word avatar is: "an intelligent entity playing a part in a game".

2.3.6 AI Middleware and Dedicated Hardware

While most game AI solutions are proprietary there are several game AI techniques that are frequently used in a variety of games. Consequently there have been a number of attempts to create game AI SDKs for generic solutions to these common problems [Fairclough et al. 2001]. So far this kind of middleware has followed rather than led the development of game AI. Mainstream games apply innovative designs a long time before they appear in middleware solutions. As a result these SDKs have found limited acceptance in the games industry [Skibak and Stahl 2002] and although there is a growing market within the game development community, AI middleware is still looked at with a considerable amount of suspicion [Dybsand 2003] with only a few solutions finding widespread use.

The AI middleware solutions that are currently available are not necessarily bound to the field of computer games and as a result the AI techniques they implement differ from product to product. Some have originally been created as 3rd party extensions to 3D animation software; others were developed for military simulation purposes. The interfaces that they provide vary greatly from code centric APIs for programmers to complex GUIs (graphical user interface) for designers. As such each system is relatively task-specific which makes these systems useful for some tasks but unable to carry out others. The greatest problem faced by the creators of the middleware is a lack of standard interfaces. Game AI interface standardisation would provide common ground for developers and middleware based on those interfaces should find easier acceptance from the industry. To that end the IGDA AI Interface Standards Committee is currently attempting to formalise the use of game AI [Nareyek et al. 2005].

Some game AI researchers are convinced that at some point in the future dedicated hardware for game AI, co-processors similar to the GPUs that have revolutionised graphics in computer games, will become available [Funge 1999]. The main hindrance for this type of hardware is the lack of a market, as chances are that the only use for this specialised and therefore expensive kind of hardware

would be computer games. The target audience for this kind of equipment would be hard-core game players, who make up only a fraction of the total number of computer game players, making the investment of time and resources in the research and development of dedicated hardware for games largely uneconomical. However that does not mean that there won't be any hardware solution for computer game AI. Using GPGPU (general purpose GPU) computation techniques, some AI calculations are already carried out outside the main processor and on GPUs instead [Erra et al. 2004]. Furthermore, only recently co-processors for physics and dynamics simulation [Hegde 2005] for use with games were introduced, providing further computing power that could be used for AI calculations themselves or to free up CPU resources for AI. Finally, the introduction of multiple-core CPUs provides developers with what amounts to a generic programmable co-processor that could be adaptable to a number of different problems, including AI, physics and graphics, as can be seen in recent games console developments [Reynolds 2006].

Chapter 3

Data-Driven Architecture in Computer Games

Data-driven design takes program modularisation and code-reusability to its extremes. It is the logical progression from separating out task-specific functionalities into distinct APIs and the use of common application frameworks to speed up program development.

3.1 Data-Driven Design

In software development, in general, the use of a data-driven architecture usually means the distinction of an application's core components from application specific code. The former are code elements that may be reused unchanged in other applications, whereas the latter indicates code or data that is unique to the individual application. This implies an abstraction of the application's internal logic from the data which is used to define the application's behaviour [Rabin 2000c], separating the definition of the application's make-up from the application's core functionality, which becomes effectively "policy free".

Being "policy free" means that while the application's core provides functionality which entails only the means for the creation of an application, i.e. the building blocks from which a comprehensive application can be constructed, it does not, however provide the application's functionality itself. In simple terms, it provides the "how to do", but not the "what to do".

3.2 Data-Driven Design in Computer Games

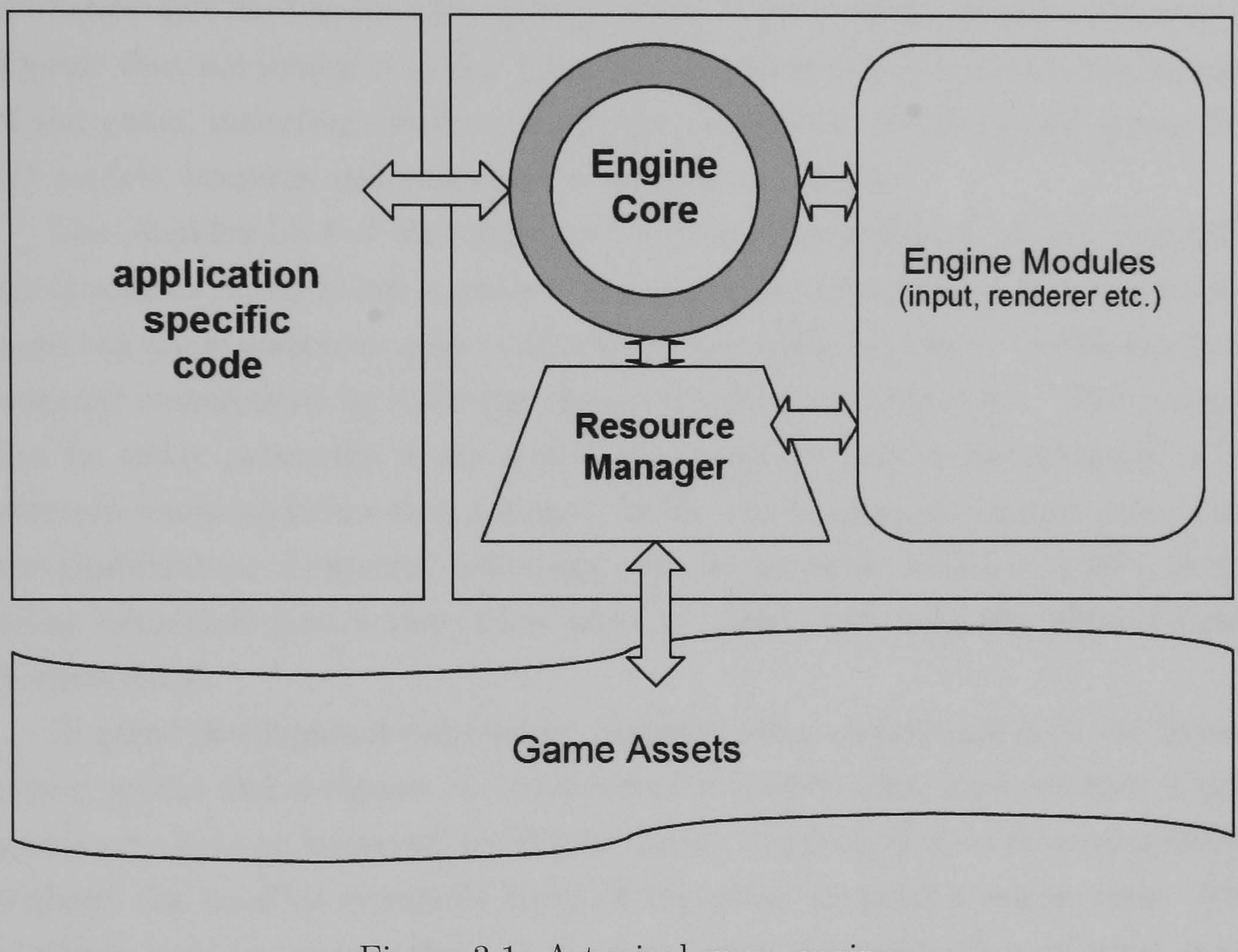


Figure 3.1: A typical game engine.

If the application is a computer game, a data-driven architecture results in games driven by a game engine [BinSubaih et al. 2007] (see Figure 3.1). This allows developers to make a clear distinction between engine (code) and game code, the former being the core elements that may be shared among several distinct games and the latter being the code that is unique to the specific game. As most of the game specific logic is no longer an intrinsic part of the core source code, in general a data-driven game engine is highly reusable and believed to be cost efficient [Danc 2006], enjoying a relatively long shelf-life.

There are different layers of abstraction that define the make-up of the data part of data-driven games, but borders between these layers are not strictly defined and vary depending on the individual implementation. In its simplest form,

the game specific data can take the form of source code which can be linked with the game engine core. A higher level of abstraction on the other end of the scale is to store this data as an external game asset. Game assets are those elements of a game that are loaded into the game engine at run-time to provide the content of the game, including elements which are created by designers and artists like 3D models, textures and animation or sounds and music.

The Achilles heel of this high level of data-driven design in any computer application is the fact that an outsourcing of product specific data into an external asset can allow malicious users to effectively hijack the system by modifying those external resources or by replacing them with their own resources. This however can be easily prevented if the application properly verifies the integrity of its external resources before they are used. In the case of computer games, sometimes the modification of external assets can even be desirable, which is evident in the many extensible games that allow users to make their own modifications (see Section 3.2.1).

In game development data-driven design is often understood as a way to empower artists and designers to independently modify game logic without a programmer's help or intervention [Wilson 2002], requiring this to be accomplished without the need to recompile parts of the game program's source code. The methods used to achieve this are the same ones that also allow external game modification.

3.2.1 Game Extensibility and Modification

Over the past decade there have been many games that have been created in a way that allows the players to directly modify the games. This "modding" of games [Wallis 2007] goes from the simple extension and addition to existing games up to the creation of completely new games. This has been supported by the games industry through the publication of the same tools used by the game designers for the creation of the games themselves. By exposing the end-user, i.e. the players, to the tools allowing them to extend and modify the games themselves and by assisting them with any game modifications they intend to make. the developers add value to a game and dramatically increase its shelf-life. To simplify this,

some games provide extensive software interfaces into the game engine, allowing parts of the games to be reprogrammed by direct manipulation of the game code or through plug-ins, however, the method by which the extensibility of most modern games is realised is by the use of more or less complex scripting systems (see Section 3.2.2 and Chapter 6).

3.2.2 Scripting and Data-Driven Design in Computer Games

A scripting system in which the script has complete control over the behaviour of the application that it is embedded in is the ultimate implementation of a data-driven design.

Varanese [Varanese 2003] explains and discusses in detail how scripting is used in combination with computer games and how scripting systems can be embedded within computer games. Scripting can be used to issue commands to the game engine, such as loading of objects, textures and levels, but also for much more complicated tasks like playing animated cut-scenes, directing camera movements or triggering events inside the game worlds. It removes a large part of the – previously hard-coded – internal game logic from the game engine and transforms it into a game asset. Scripts themselves can be used to direct the application of these assets to the game, effectively modifying the behaviour of the game engine and the game itself without the need for the game source code to be recompiled. With scripts themselves being game content, this means that the game engine only provides a shell, i.e. a protected "sandbox" environment for scripts within the game engine. Scripts operate within this "sandbox" with the scripts creating the game and its environment without being able to adversely affect the running of the game engine itself.

A number of games have built-in dedicated scripting languages, like Quake which includes a scripting language called QuakeC [Simpson 2002] or Unreal which has a scripting system called UnrealScript [BinSubaih et al. 2007], both allowing extensive modification of the games through scripting alone. Other games use existing scripting systems that have been modified according to the

game's requirements. A much more in-depth discussion of scripting languages and games is presented in chapter 6.

Chapter 4

Common Approaches to the Implementation of NPCs

NPCs are a significant factor in the playability of computer games, i.e. if the NPCs do not perform (act) as expected, the player's enjoyment of the game suffers. It therefore does not come as a surprise that with the intention to avoid unnecessary risks in NPC development, game developers have a tendency to employ proven solutions to the challenges faced by NPCs. These solutions typically include methods such as NPC behaviour definition using FSMs and the use of the A* algorithm for path planning [Orkin 2004b].

4.1 General NPC Implementation

Combs and Ardoint [2004] state that a popular method for the implementation of game AI is the use of an "environment-based programming style", i.e. the creation of the virtual game world followed by the association of AI code with the game world and the entities that exist in it. This means that the NPC intelligence is built around and is intrinsically linked to the virtual game environment. This type of NPC intelligence can be created using "traditional" methods for "decision making", "path finding" (planning) and "steering" (motion control). As mentioned before (see Chapter 2, Section 2.2.2), these are the tasks that are carried out by NPCs in most modern computer games and to which, by convention rather than technology, the actions of NPCs are usually restricted. In terms of the

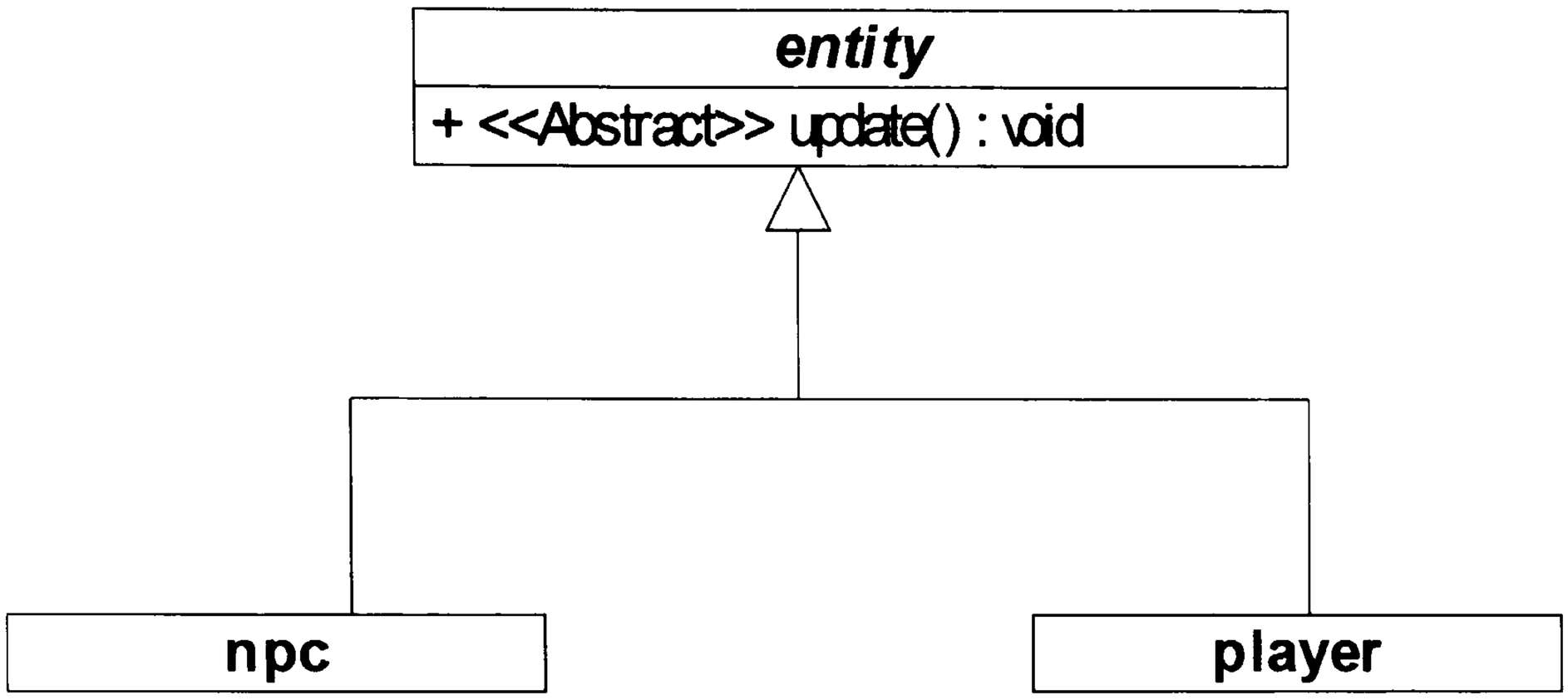


Figure 4.1: Typical entity class hierarchy in a computer game.

"perceive-think-act cycle" (see Chapter 2, Section 2.2.2.1) of human-like NPCs some of these tasks closely mirror those that have to be performed by human players, i.e. both NPCs and humans need to perceive the environment, process that information and act on it. The human player usually visually perceives the virtual world through the computer's screen, while the NPC is anchored in the virtual environment, perceiving it through sensor functions. The human player's thinking is mirrored in the NPC's decision making and path planning. The actions of human players as well as NPCs both directly affect the virtual world, so the obvious solution is to use the same interface for both, allowing them to share some of the required functionality. In games that are programmed using object orientation in the C++ programming language this can be achieved by deriving both, NPC as well as human player controls from the same base class (see Figure 4.1). This mechanism aims to allow human players and NPCs to compete on an even playing field. This is important to preserve the player's suspension of disbelief and create an enjoyable experience, as the player's enjoyment of the game would suffer if NPCs appeared to be too "stupid" or if they displayed superhuman competence at playing the game.

4.2 Decision Making

Of the three common NPC tasks, "decision making" most strongly implies the use of intelligence. In the case of the human player this usually means the evaluation of the visual information, received as input from the computer screen, which will determine the player's actions and which needs to be emulated by the NPC. The creation of a seemingly intelligent and therefore believable NPC requires the formulation of rules to govern the NPC's behaviour, allowing the NPC to perceive and interact with its environment. To formalise this, Funge [1999] applies the following equation:

$$\textbf{behaviour} = (\text{domain})\ \textbf{knowledge} + \textbf{instruction}$$

Funge's definition of "instruction" encompasses pre-defined rule based behaviour for NPCs (see Chapter 2, Section 2.3.1). His definition of "domain knowledge" includes information that allows an NPC to take reasonable decisions, such as axioms describing cause and effect of actions that allow NPCs to develop action plans to achieve nondeterministic, goal-directed behaviour. This combines deterministic and nondeterministic behaviour methods to create seemingly intelligent NPCs that can dynamically decide on actions but who also always have a fall-back position in case the NPC's plan fails. Funge's definitions can be extended, however, if one assumes the NPCs behaviour to be mainly reactive, i.e. directed by events that occur in the virtual world. In this case one could refer to the NPC's behaviour as instinctive behaviour. Approached from this ethological point of view, we have defined the domain knowledge of an NPC as follows [Zerbst et al. 2003]:

$$(\text{domain})\ \textbf{knowledge} = \textbf{instincts} + \textbf{perception}$$

The instincts are the rules that define the NPC's reactions to stimuli (sensor data) from its environment, making them effectively low-level instructions for the NPC. They are directly dependent on actual perception of the virtual game world at a given moment in time, i.e. the inputs received from the NPC's sensors, and combined with the latter these rules provide the NPC's domain knowledge. While the game is running this domain knowledge is evaluated during the NPC's

decision making process and then augmented with the pre-defined instructions to produce the NPC's actual behaviour.

Funge's equation is inclusive, allowing for rule-based techniques, as well as knowledge-based and machine intelligence methods. Only a small minority of games perform decision making by employing machine intelligence techniques, such as neural networks that have been trained to select appropriate reactions for situations that arise in the game world. For these on-line learning has usually been disabled as this method's outcome is hard to predict and may therefore have a negative impact on the "game experience" if NPCs learn undesirable behaviours. Consequently in most commercial games decision making is implemented using more or less complex finite state machines.

4.2.1 Implementation of Finite State Machines for NPC Behaviour

FSMs in game development are more flexible than the formal definition for deterministic FSMs in computer science that usually have only single states follow one another, whereas the loose definition in games allow each state to have several possible follow states. In game FSMs each state is usually associated with a specific behaviour and an NPC's actions are often implemented by linking behaviours with pre-defined animation cycles for the NPC that allow it to enact the selected behaviour [Orkin 2006].

A typical scenario found in many computer games that would use an FSM involves NPCs on patrol, guarding an area in the virtual game world. These NPCs will follow a pre-defined path on their patrol and react to disturbances caused by other NPCs or human players entering the area they are guarding. This type of scenario could just as well exist in RPGs[1], as in FPS or RTS games, making it a suitable model for further examination. An example state machine for this game scenario could hold the states 'patrolling', 'challenging intruder'

[1] An RPG or Role Playing Game belongs to a computer game genre that has been derived from traditional paper-based games and board games like the popular "Dungeons and Dragons". In these games the player usually controls a hero character or a party of hero characters and needs to solve a series of quests within a fantasy setting.

and 'attacking intruder' (see Figure 4.2). The first of these states is the NPC's default state for the 'patrol' behaviour that is executed by the NPC when no other entity is within its patrol area. The second state is entered when an entity enters the NPCs patrol area, resulting in the execution of the 'challenge intruder' behaviour. If that entity is identified as friendly, the NPC reverts back to the 'patrolling' state; however, if the entity is identified as hostile, the third state is entered and the 'attack intruder' behaviour is executed.

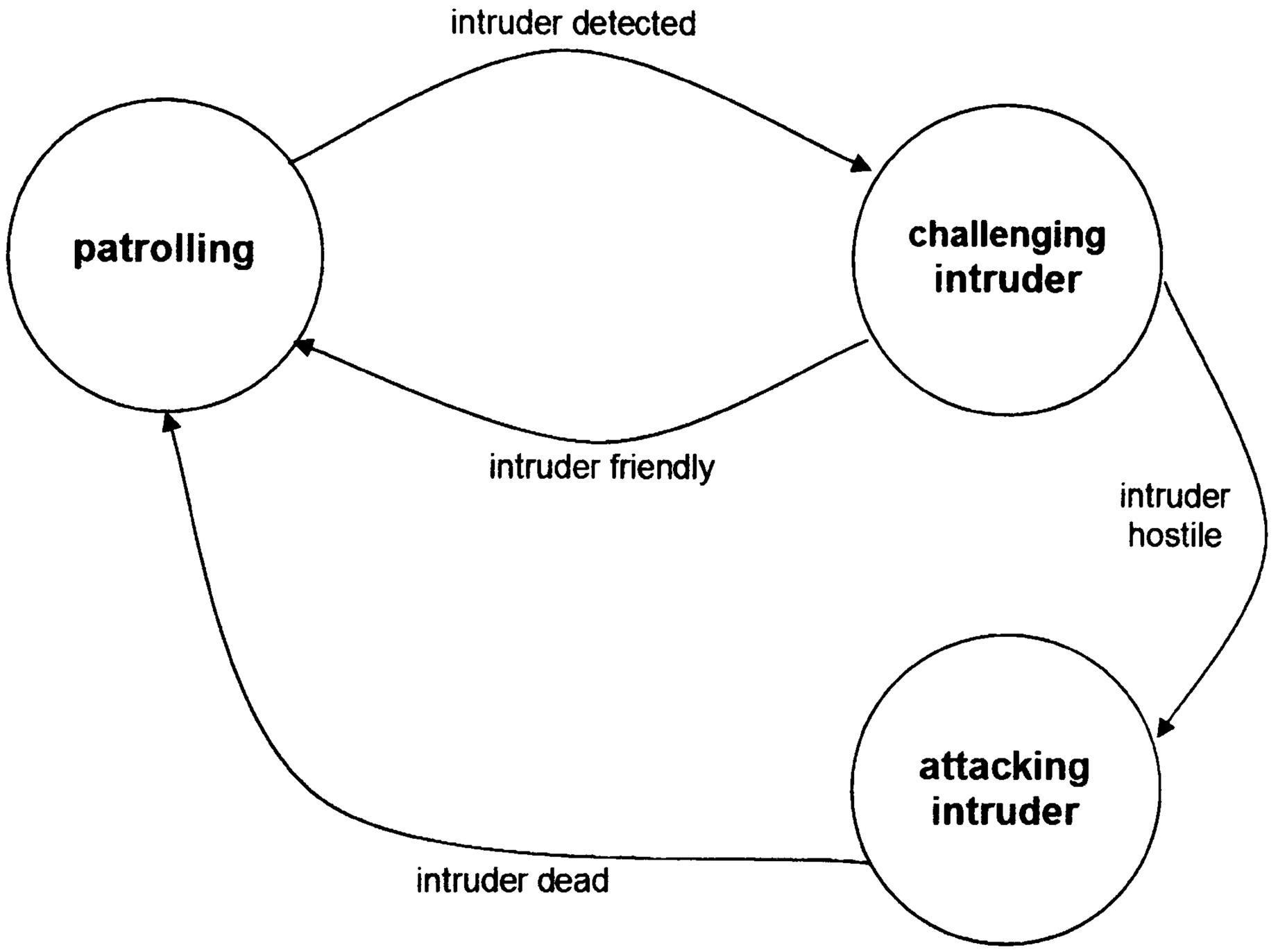


Figure 4.2: Finite State Machine for a typical NPC.

In its simplest form the implementation of a finite state machine in a computer game will take the form of a multiple selection in which each case represents one of the states of the FSM. This is then evaluated once during each execution cycle for this NPC to determine if the current state needs to change or to execute any actions that need to be performed for the current state. The FSM for the game

scenario described above could then be implemented as follows:

```
enum {patrolling, challenging_intruder, attacking_intruder} state;
...
switch(state)
{
  case patrolling:
  if(intruder_detected()) state = challenging_intruder;
  /* execute 'patrolling' behaviour */
  ...
  break;
  case challenging_intruder:
  if(intruder_hostile()) state = attacking_intruder;
  else if(intruder_friendly()) state = patrolling;
  /* execute 'challenging_intruder' behaviour */
  ...
  break;
  case attacking_intruder:
  if(intruder_dead()) state = patrolling;
  /* execute 'attack intruder' behaviour */
  ...
  break;
}
```

If implemented in C/C++ as above, this code can be problematic due to the peculiarities of the mechanics of the multiple selection available in C/C++ ('switch' statement) in which there is a fall-through between the different cases. This makes the implementation error prone, as easily missed logical errors can cause unwanted side effects. For instance, omitting a single 'break' instruction between cases that then lead to unexpected results are hard to debug, as they are syntactically correct. An alternative to this type of implementation would use a series of nested dyadic (if-else) selections as the listing below demonstrates:

```
if(state==patrolling)
{
  if(intruder_detected()) state = challenging_intruder;
  /* execute 'patrolling' behaviour */
  ...
}
else if(state==challenging_intruder)
{
  if(intruder_hostile()) state = attacking_intruder;
  else if(intruder_friendly()) state = patrolling;
  /* execute 'challenging_intruder' behaviour */
  ...
}
else if(state==attacking_intruder)
{
  if(intruder_dead()) state = patrolling;
  /* execute 'attack intruder' behaviour */
  ...
}
```

While using these dyadic selections would avoid the problems caused by unwanted fall-throughs, possible errors here would be the accidental use of a monadic selection, effectively breaking the structure of the FSM by possibly allowing several states to be entered or even the wrong states to be evaluated during a single execution cycle. The creation of an FSM using this type of dyadic selections can be simplified using a macro-based language (see Chapter 6, Section 6.1) [Rabin 2002b], which also prevents the introduction of errors into the FSM definition which may otherwise be hard to debug. Using this FSM language, the implementation of the above FSM would be easier to maintain and take the following form:

```
BeginStateMachine
  State(patrolling)
    OnUpdate
```

```
    if(intruder_detected()) state = challenging_intruder;
    /* execute 'patrolling' behaviour */
    ...
  State(challenging_intruder)
    OnUpdate
    if(intruder_hostile()) state = attacking_intruder;
    else if(intruder_friendly()) state = patrolling;
    /* execute 'challenging_intruder' behaviour */
    ...
  State(attacking_intruder)
    OnUpdate
    if(intruder_dead()) state = patrolling;
    /* execute 'attack intruder' behaviour */
    ...
EndStateMachine
```

4.2.2 Alternative FSM Implementations

Similar functionality can be achieved through object oriented methods using a state class. Whatever method is used, however, a problem that remains is that this type of FSM implementation may not scale well, i.e. it can easily grow to a size that will leave it in a confusing and therefore unmaintainable state. One possible solution to this problem is the use of a hierarchical state machine that breaks up complex states into a set of smaller ones that can be combined, allowing the creation of large and complex FSMs.

An alternative to these hard-coded solutions is the use of data-driven FSMs that only require a relatively small amount of code with the actual data contained in external assets. This allows the use of specialist tools for the construction and maintenance of the state machine (see Chapter 6, Section 6.3.4).

4.3 Path Finding

After decision making, the next task for NPCs is "path finding", i.e. the identification of a travelable route between the NPCs current position and its destination in the virtual world. In the context of the above patrolling NPC example (see Section 4.2.1), this could mean the planning of a path between the waypoints that the NPC needs to visit during its patrol as well as the generation of a path for intercepting intruders that enter the NPC's patrol area.

A path finding priority is usually the discovery of the shortest or most cost-efficient path from the NPC's current position to its desired destination. A requirement for achieving this is the calculation or estimation of the cost involved in travelling the path which is dependent on the application, i.e. there is no prescribed method for calculating this cost. Consequently, any path finding implementation in a game will have to be provided with a suitable cost evaluation method.

4.3.1 Evaluating the Cost of Travel

The most obvious measure for calculating the cost of travel is the distance between start point and destination. Other conceivable influences on the cost of travel are surface properties that could influence the NPC's progress while moving across the terrain [Stout 2000], such as surface type, texture, consistency or condition. These surface properties can be used to simulate the effects that forces, such as friction, would have on the cost of travel. From these considerations the following equation can be derived:

$$c = |\vec{D} - \vec{S}| \times \frac{p_S + p_D}{2}$$

In the above equation, c is the cost of travel; $\vec{S}$ and $\vec{D}$ are position vectors encoding start and destination positions respectively. The value p_S is a modifier encoding the start surface property and the modifier value p_D encodes the destination surface property. Consequently the cost of travel is the product of the distance between start and destination, i.e. the length of the vector spanning

from start to end point, with the arithmetic average of the sum of the surface property modifiers of the start and end points.

Additional data that could be taken into account are height differences in the virtual world's topography, i.e. upwards or downwards sloping of the terrain, which could be used to generate an additional weight value to be factored into the cost calculation. Travelling down a slope should reduce the cost of travel, whereas travelling up a rise in the terrain should increase the cost, resulting in the equation shown below:

$$c = |\vec{D} - \vec{S}| \times \frac{p_S + p_D}{2} \times h$$

The modifier value h in this equation is supposed to act as a weight encoding the height difference from start to destination. This is to provide an upwards or downwards correction of the cost of travel depending on the presence of a slope in the terrain's topography. For this the value h is defined as follows:

$$h = 1 + \frac{D_y - S_y}{5 \times |D_y - S_y|}$$

The result of this is a plausible cost equation that can be used for path planning in computer games.

4.3.2 Virtual World Representation

The virtual world in which NPCs, as well as player characters reside needs to exist in a form that can be perceived and processed by the NPCs. In many modern computer games large portions of the virtual game world are represented as a graph, i.e. as a set of interconnected nodes that encode the area which the NPC can traverse. These nodes are sometimes generated from a so-called "Navigation Mesh" [Snook 2000], "a set of convex polygons that describe the 'walkable' surface of a 3D environment" [Tozour 2002a]. This can be derived from the world's actual geometry by simplifying it to form a mesh that encodes the world's geometric extremes.

The minimum information required for a node in a graph that defines the search space for path finding are the node's position in the virtual world, as well

as its connections to other nodes. This can then be augmented with additional data, such as information regarding the world's surface properties at the node's position. Within a computer game, the node information would usually be stored inside a record data structure.

An implementation of a node record encoding the node's position as a point in space given by its Cartesian co-ordinates, annotated with an additional value that holds information about the surface properties and storing the node's connections as a null-terminated array of links to neighbouring nodes, could take the following form:

```
struct node
{
  double x;
  double y;
  double z;
  double p;
  struct node **neighbours;
};
```

A cost function using the above cost equation (see Section 4.3.1), using the straight-line distance between two nodes and taking into account surface properties, as well as the virtual world map's topography, could be written as shown below:

```
double cost(node *s,node *d)
{
    double h = 1.0;
    double x = (d->x - s->x)*(d->x - s->x);
    double y = (d->y - s->y)*(d->y - s->y);
    double z = (d->z - s->z)*(d->z - s->z);
    double c = sqrt(x+y+z);
    c *= (s->p+d->p)/2.0;
    h += (d->y - s->y) / (5.0*fabs(d->y - s->y));
```

```
    c *= h ;
    return c;
}
```

The encoding of the virtual world that uses a graph of nodes using a data structure, such as the above node record structure, as well as the provision of a cost of travel function, such as the function shown above, are the requirements for the implementation of path finding in the virtual world. To then search the virtual world for the shortest path between two locations within it, a planning algorithm must be applied to the information provided in the graph.

4.3.3 Planning the Path

The most popular path planning algorithm used in modern computer games is the A* algorithm [Matthews 2002] (see also Chapter 2, Section 2.3.2.1), a generalisation of Dijkstra's algorithm [Dijkstra 1959], which is guaranteed to find the least costly path if such a solution exists within the search space.

A* performs an iterative best first search of its search space using a heuristic based on three functions:

1. $g(x)$, named "goal", i.e. the actual cost involved in reaching the current node from the start node

2. $h(x)$, named "heuristic", i.e. an estimated distance to the destination node from the current node that should be an underestimate of the actual cost for the algorithm to find the optimal solution [Dechter and Pearl 1985]

3. $f(x)$, named "fitness", i.e. the sum of the functions $g(x)$ and $h(x)$, resulting in an estimated cost for the path from the start node to the destination node

Beginning with the start node, the results of the evaluation of these functions informs the selection of the next node from the search space to be examined. To achieve this, the algorithm requires additional information to that stored within the nodes of the graph that defines its search space, i.e. fitness, goal and heuristic values as well as links to the actual node data structure, as well as the parent node:

```
struct pathnode
{
  node  *mapnode;    // node within the graph
  double fitness;    // sum of the goal and heuristic values
  double goal;       // cost of travel up to current node
  double heuristic; // estimated cost of travel to destination
  pathnode *parent; // parent node within the path
};
```

The planning algorithm (see Appendix A for an A* sample implementation) returns a list storing the nodes of the path from start node to destination node, which can then be used as waypoints by the NPC.

4.4 Steering

Once a path from the NPC's position to its destination in the virtual world has been discovered, the final task that an NPC needs to accomplish is to move to its destination in a believable manner. This is achieved using "steering", i.e. navigation and motion control. This incorporates several methods of varying complexity, ranging from totally random movement via exploratory terrain traversal in unknown environments, which is unplanned as there is no known destination, to the rigid following of a given path. "Path following" is a "steering behaviour" [Reynolds 1999] that involves following a pre-planned path through the virtual entity's environment. These generated paths are frequently unsuitable for creating believable NPC motion, a common problem that stems from the fact that paths in virtual environments usually take the form of straight line segments connecting the nodes that make up the path. The simplest solution for creating smooth appearing movement along the path over time is the application of an interpolating parametric curve (spline) through the path's nodes [Rabin 2000a] and to follow the resulting curve rather than straight lines between the nodes.

Improvements that can be made to generate better believable NPC movement are the addition of other steering behaviours, such as "local steering" methods

[Tomlinson 2004] that facilitate (dynamic) obstacle avoidance, allowing the NPC to exist in a dynamic, changing environment without the need to constantly re-evaluate its planned path. These "local steering" methods also include emergent behaviour methods such as flocking [Reynolds 1987] (see also Chapter 2. Section 2.1.2), which are useful in situations when pre-computed plans do not exist or fail due to unforeseen changes to the virtual environment.

4.5 Construction of an NPC

The typical techniques for implementing NPCs described in this chapter are commonplace in most computer games and can be employed to construct NPCs with or without the use of a data-driven architecture (see Chapter 3), which could tie together the individual components that make up the NPC.

As stated above (see Section 4.1), in a game application the code performing the implementation of NPCs usually shares a lot of code with that of human players' avatars. Code elements that are often identical for both (NPC and human player's avatar) are the methods that encode effectors for actions that can be carried out by both.

Methods that are usually unique to the NPC object are the sensor functions that enable it to perceive the virtual world. A method that NPC and human player's avatar have in common, however, is an 'update' method which updates the entity's state and position in the virtual world and which is usually called for each update cycle of the application (once every frame). In addition to the instructions found in the 'update' method of a human player's avatar, an NPC object's 'update' method usually also incorporates the NPC's "perceive-think-act cycle".

In most game applications the "decision making" process is unique for each NPC type and sometimes even for each NPC in the game. Decision making code that can incorporate FSMs such as those described above (see Section 4.2.1) and which may include the initiation of the execution of actions (equivalent to the human player's input to an avatar) is either placed inside the 'update' method for each NPC object or within a separate method which is called from the 'update' method.

Path planning, which is generally initiated during the "decision making" process for NPCs (but is also used for indirectly controlled avatars of human players), is usually implemented as a generic method which is identical for all NPCs (frequently implemented within a top-level class), whereas cost calculations may vary from NPC to NPC as different types of NPC might be influenced by the virtual world's terrain and its properties in different ways, requiring each NPC to supply their own cost of travel function to the planner. A further reason for decoupling the cost calculation from the planner is that some NPC implementations will keep the planning method even more generic, allowing it to be used for other purposes than just path finding [Higgins 2002a] (see also Chapter 2, Section 2.3.2.2). Finally, "steering" is a mostly generic task that is usually identical for all NPCs and therefore usually implemented as a method within a top-level class, unless steering behaviours that are specific to a type of NPC are used. which would require the NPC object to supply their own steering methods.

Part II

Syntactic Behaviour Definition

Chapter 5

NPC Behaviour Definition Languages for Computer Games

While the use of scripting in games can mean simple manipulation of the appearance of the virtual game environment, one of the main areas in which games allow modifications of this kind is in the behaviour of the game AI. Furthermore, the use of scripting is also the most common method by which the AI behaviour of a game is extended or modified. In fact, one of the features that people have come to expect when it comes to FPS games is the provision of a scriptable interface for controlling NPCs. In the context of NPCs, simple data-driven design in which the behaviour of the AI entities in a game depend on the interpretation of an external game resource (i.e. a script program) effectively bridges the gap between hard-coded AI and fully scripted NPCs.

5.1 Behaviour Definition Languages

We define the term Behaviour Definition Language (BDL) – not to be confused with the term "Behaviour Description Language" [Bertrand and Augeraud 1999] – to be a programming language used for the definition of game character behaviour (in the ethological sense of the word), often found in the form of programs running on a virtual machine which interfaces with the character controls within the game engine. Thus the task of BDLs is to facilitate the application oriented creation of believable virtual entities that inhabit game worlds. While behaviour definition

languages are domain specific to the creation of NPC intelligence, they are often more than just a Domain-Specific Language (DSL) [West 2007] for game AI. Many behaviour definition languages maintain the flexibility of traditional programming languages while at the same time offering powerful AI functions and operators.

As the purpose of BDLs is to facilitate the definition of artificially intelligent behaviour, it may be beneficial for the design of such languages to utilise elements and concepts found in AI languages (see Section 5.1.1). BDLs also bear some considerable resemblance to the educational mini-languages [Anderson 2004] that have found use in computer science education for decades [Brusilovsky et al. 1997] (see Section 5.3.3). These mini-languages usually provide a task-specific set of instructions and (sensor) queries which allow users to take control of virtual entities or actors, acting within a micro world, similar to a BDL, controlling an NPC that inhabits a virtual game world.

5.1.1 AI Languages

There exist a number of languages that were designed with AI applications in mind, some of which could be categorised as behaviour definition languages, but most of these languages are unsuitable for direct application to NPC behaviour definition in computer games. Some rule-based AI languages provide hybrid programming methodologies that combine elements of logic programming languages (from traditional AI research) and the more commonly used imperative implementation languages [Wright and Marshall 2000]. These implementation languages provide the most often used means for NPC behaviour definition in computer games, and even in the simplest form they can be used successfully in that capacity. One example for this is the use of the AWK [Aho et al. 1979] based GAWK (Gnu AWK), a simple scripting language, which has been used as an AI problem solving language to great success, in some situations attaining better results than those achieved with traditional AI languages [Loui 1996].

Programming languages that have been developed especially to solve problems in the development of AI, such as LISP [McCarthy 1959] or logic languages such as Prolog, which use a declarative paradigm for the definition of a search space in which a solution for the problem may be found rather than an algorithm that

describes the solution to the problem [Colmerauer and Roussel 1993], are too different from the C-like procedural languages, which allow a simple mapping from the way that algorithms would be expressed in natural languages to the programs. The use of these AI programming languages would be considerably more difficult for a non-programmer to learn than the use of a procedural language and consequently complicate the definition of the behaviour of virtual entities.

However, this does not mean that it is impossible to use these AI programming languages in the context of computer games. We have used a LISP based language (GP Asteroids Script) for defining the behaviour of an artificial player in an arcade game [Anderson 2002] (see Chapter 7, Section 7.1), and another LISP based language, Tapir [King et al. 2002], has been used to define AI entities in war-games. The target user group for Tapir, however, is a programmer who understands the difficulties of agent control, thus making it unsuitable for non-programmers. Tapir is nonetheless one example for the number of AI specific languages that can be used to solve the kind of problems faced by NPCs in computer games. Some of these languages have their origins in the field of cognitive robotics, a selection of which is presented below, and one of the attributes that many of them share is the concept of "Action Languages" [Gelfond and Lifschitz 1998]. Gelfond and Lifschitz describe action languages as a formalized method for describing the cause and effect of actions within an environment, a domain that robotics shares with AI entities in a virtual game world. They differentiate between two distinct categories of action languages that can also be seen as individual components that can be combined into a unified action language [Lifschitz 1997]. Those two components are action description languages, used to express the rules that define state transition systems (a category that is matched by several logic programming languages), and action query languages, which can be used to express "properties of paths in a given transition system" [Lifschitz 1997].

The "classical" AI languages, i.e. those that are not concerned with NPCs in computer games, can often be found among (but are not restricted to) constraint logic languages and cognitive modelling languages for goal-oriented systems (a recent development of which are the high level behaviour representation languages

that aim to simplify access to intelligent systems and make them easier to use and comprehend [Ritter et al. 2006]).

Other AI languages that can be applied to NPC creation are agent oriented languages [Huget 2002], as most AI entities in computer games can be classified as agent programs. Agent oriented languages often bear similarities to object oriented languages [DeLoach 1999] and programs developed using these languages can often be represented by a visual abstraction, allowing the use of meta languages like UML to be used for defining the agents (see Chapter 6, Section 6.3.4). This can considerably simplify the AI entity development process and makes this kind of language ideal for the definition of NPC behaviour in computer games.

While many of these languages are used to direct the behaviour of artificial entities, i.e. robots (physical and virtual), not all conform to our definition of behaviour definition languages. Instead they are modelling languages that aim to indirectly describe human-like behaviour to be realised by an underlying architecture, rather than the behaviour definition (programming) languages that describe an algorithm that creates the illusion of human-like behaviour. While the effect may be similar, the methods used to generate the entities' behaviour are vastly different. Behaviour definition languages provide direct control of the behaviour, whereas in the case of modelling languages, the behaviour is emergent and beyond the developer's immediate control.

5.1.1.1 GOLOG – A Cognitive Robot Control Language

GOLOG (alGOL in LOGic) [Levesque et al. 1997] is an action language developed by the cognitive robotics group at the University of Toronto for the purpose of behaviour definition for robots. While GOLOG is at its core a logic programming language that is based on the situation calculus [Levesque et al. 1998], it explicitly provides high-level control operations for robots that allow the definition of action sequences for execution by the robot, as well as program flow control structures, such as sequence, selection and iteration which are more reminiscent of imperative programming languages, allowing a blend between logic and imperative programming styles.

The lack of immediate feedback and low-level control in GOLOG which does not provide any fault tolerance or means to handle run-time errors has prompted the development of the "execution and monitoring system" GOLEX [Hähnel et al. 1998]. GOLEX is a companion system for GOLOG which resides in-between the high-level GOLOG and low level control software, extending GOLOG with the means for implementing simple interaction and sensing in GOLOG programs.

5.1.1.2 GRL – A Language for Robots and Game-Bots

The "Generic Robot Language" (GRL) is a functional language for the definition of behaviour-based systems [Horswill 2000], which is an extension to the high-level programming language Scheme [Steele and Gabriel 1993], a functional language that is itself based on LISP [McCarthy 1959]. The language itself, extendable through macros, only exposes (makes accessible) a sub-set of its host language, requiring the use of GRL for most tasks and restricting the use of Scheme program code to the definition of signal sources as well as the expression of some form of FSMs.

GRL was originally developed to write control programs in behaviour-based robotics, defining robot behaviours at a relatively low level at the expense of the expressiveness that could be achieved with a language such as GOLOG (see above), however the resulting performance gain, combined with the functionality provided by GRL, lends itself perfectly to the definition of game character behaviour. For this, the language provides the means to implement and easily combine higher level operators, allowing the creation of concise yet powerful behaviour definition programs.

The entities controlled by GRL programs reside in an event driven environment which needs to be provided by the underlying architecture, with GRL programs continuously processing the signals they receive. The language is not bound to a specific robot architecture and can output programs for use with a variety of systems, one type of which are programs in the UnrealScript language that can be used to control NPC behaviour.

The use of GRL in conjunction with FlexBot [Khoo and Zubek 2002], a software development kit for game-bots (NPCs) for the commercial game Half-Life,

has shown the system's capability to enable the parallel existence of a large number of very convincing NPCs within a typical game environment. The game-bots resulting from the application of this approach (see Chapter 2, Section 2.2.2.2) have complex state machines at the heart of their behaviour which are of a similar kind to those that have previously been defined manually for NPC behaviour in commercial computer games.

5.1.1.3 CML – Cognitive Modelling for Animation

John Funge's Cognitive Modeling Language CML [Funge 1998] is a high level behaviour definition language for AI entities in computer games and computer animation. Funge created CML to provide an intuitive method for creating virtual entities that have the ability to interact with the virtual world that they inhabit. CML is related to the programming language GOLOG [Funge 1999] (see Section 5.1.1.1 above) in so far as like GOLOG it is based on the situation calculus [Levesque et al. 1998]. CML, however, was designed with computer games and computer animation in mind. CML aims to strike a balance between cognitive modelling and deterministic methods by providing means to use both, employing deterministic techniques as fall-backs for the nondeterministic methods. CML uses the situation calculus to provide the NPCs with the necessary domain knowledge to help them understand their environment and their own situation within that environment by defining preconditions and by expressing the effects that NPC actions will result in within the game world. This description can then be interpreted as the desired behaviour by a run-time system. The situation calculus is used to define a world by describing world states and the possible combinations of actions that can lead to the creation of these states which is a similar concept to that of the action languages described above (Section 5.1.1). The precise relationship between the situation calculus and action languages and methods for translating expressions from one to the other are described by Giunchiglia and Lifschitz [Giunchiglia and Lifschitz 1999]. The syntax of CML is based on the mathematical notation of the situation calculus but is held close to the English language to simplify program development.

5.1.1.4 Soar and Related Systems

Soar (see Chapter 2, Section 2.2.2.1) is a software toolkit used in AI research. It includes an AI programming language as well as an architectural framework for creating autonomous agents with human-like cognitive abilities. The AI language allows the description of production rules that are stored in a knowledge base in the memory of the Soar framework. The production rules map conditions (states) to actions and the knowledge base of productions provides a search space from which the behaviour of an artificial entity can be selected. An agent using this framework gathers world state information (from sensor data) as its inputs and searches the productions in its knowledge base for the most appropriate action which is then passed as output to the environment. If no appropriate solution is found in the search space, i.e. if the agent cannot decide what to do, a machine learning mechanism in Soar attempts to develop an alternative solution through the automatic generation of additional productions.

Soar programs themselves provide a relatively low-level of abstraction, i.e. the Soar language cannot really be counted as a high-level BDL if compared to other AI languages. The real power of the system lies in its architecture, which is targetable by higher level AI programming languages such as the high level behaviour representation language Herbal [Cohen et al. 2005].

5.2 Requirements for Behaviour Definition Languages

The design of a programming language for the definition of artificial behaviour as an extension to a specific game or genre of computer games (for example First Person Shooter games) is relatively simple if only deterministic behaviour is involved. For instance, the first prototype for our ZBL/0 behaviour definition system [Anderson 2004] (see Chapter 7, Section 7.2) – an educational tool for learning how to syntactically define NPC behaviour in FPS (First Person Shooter) games [Zerbst et al. 2003] – was developed over a period of little more than a fortnight (from conception to first use). In effect such a language does little more than provide a function binding interface to a game engine for the creation of

rule based systems. The game engine itself does all the work while the script program only ties together the different game engine components that provide the NPCs with functionality. Unfortunately the specialisation for a single genre greatly restricts the reusability of such systems and they are usually proprietary to a specific product or range of products.

5.2.1 Language Requirements

A system that controls the behaviour of autonomous agents in a virtual game world usually exists on two levels [Anderson 2004]. The higher level is a behaviour definition (scripting) language that often resembles a traditional programming language, whereas the lower level is the corresponding run-time engine which interfaces with the game (see Section 5.2.2). The former, i.e. the BDL, needs to achieve a number of objectives. Some of these objectives are conflicting, so compromises will need to be found. For computer game developers to benefit from a BDL, it has to be designed to be intuitive (see Section 5.3.1), i.e. the language must be easy to learn and possibly easier to use than traditional programming languages. One way this could be achieved would be by making the language as similar to a natural language as possible, as suggested by Funge [1998]. It is our belief. however, that a close resemblance of a behaviour definition programming language to a natural language may easily prove counterproductive (see Section 5.3.2). We are also convinced that the notion that a traditional programming language may be too complex for non-programmers to use is wrong. A much more practical approach would be to base a BDL on an existing production language (see Section 5.3.4). Furthermore, a BDL should not only be intuitive, but it should also be kept as generic as possible to be useful for the creation of computer games of different genres. While the generation of simple deterministic behaviour for NPCs may be suitable to some games, other games may require their entities to have goal-directed behaviour. Consequently, both of these AI methods will need to be accommodated by the language.

Among all possible programming language features, we have identified the following to be especially useful for BDLs:

- a state machine data type (finite and possibly also fuzzy – see Chapter 2, Section 2.3.1)
- entity annotation and smart environments (see Chapter 2, Section 2.3.3.4)
- goal-orientation specific data types and operators (see Chapter 2, Section 2.3.2.2)
- simple object orientation (because an NPC entity is analogous to an object [DeLoach 1999])

A programming language for the definition of NPC behaviour should therefore incorporate as many of these elements as possible while avoiding any impediment of the system's ability to direct NPCs in real-time games. While some of these features, such as special operators, can be addressed by direct integration into the BDL, others might preferably be implemented as functions of a standard function library to accompany the BDL and not within the confines of the core language itself. Similar to the C/C++ programming languages, the use of intrinsic functions within the definition of the BDL that would be hard-coded into the run-time system should ideally be avoided, i.e. the language core as such should not provide the system with any specific functionality. Instead, all functionality for the AI definition with the BDL should be provided through external functions which would be implemented as libraries for optional inclusion into programs. The minimum functionality for defining artificially intelligent NPCs using the BDL should be provided in the form of a standard library containing standard functions and compound data types. The functions provided by this standard library must enable a user to define an AI entity's domain knowledge, i.e. to anchor the NPC's perception of its virtual environment to its understanding of that environment. For the benefit of upwards compatibility to future developments, a standard library should also provide the BDL with interfaces to frequently used game AI functions as defined in the findings of the IGDA AI Standards Committee [Nareyek et al. 2004], once those interface definitions have been published. All additional functions that do not directly aid the definition of NPC behaviour but which may be useful for NPC program development should not be part of

the BDL's standard library itself. Instead those functions should be incorporated into a secondary set of utility libraries.

5.2.2 Run-Time System Requirements

The low-level run-time element of the behaviour definition system should be a scripting system, i.e. a specialised embeddable program module to execute BDL programs within the host game engine. The benefit of this is that the game application itself does not have to be recompiled for the changes to the game's NPCs to take effect. The run-time system could take the form of an interpreter which translates and executes BDL scripts during run-time. Preferably it will be a virtual machine, executing programs that have previously been translated into intermediate code, targeting the virtual machine. This translation could be done by a compiler that could be implemented externally or as an internal ahead-of-time (AOT) compiler or possibly even as an internal OTF (on-the-fly) compiler. Both forms of scripting system provide the same benefits to a game, as both allow the alteration of NPC behaviour by modifying a script program. Code contained within a BDL's libraries (see Section 5.2.1), however, should not be bytecode for the system's virtual machine or code written in the BDL, but bytecode of the native environment of the run-time system's host application for dynamic loading and execution by the virtual machine or interpreter.

The requirements for the run-time system therefore are:

- implementation as an embeddable module or as a plug-in for the host application
- independence of BDL programs from the rest of the application (to prevent run-time instabilities) and pre-emptive program termination if the environment changes beyond expected limits
- as small an overhead as possible for the execution of BDL programs
- platform independence (to the highest possible degree)

For an application's run-time stability it is very important that the virtual machine that executes a BDL's programs does so independently from the application

into which it is embedded, so it will be impossible to crash the application by executing an erroneous BDL program. In the case of an erroneous BDL program being run the virtual machine should be allowed to degrade gracefully, i.e. it should have the capability to detect the error and to stop execution of the program without interfering with its host application.

While it would be desirable for the run-time system to notify the host application of any errors that have occurred, it still must be able to act independently without requiring the host application to select the next operation. This will have to be addressed by the run-time system's API, the interface that will allow the host program access to the run-time system. This API will also need to be able to map the data and functionality of the host application to the corresponding structures within the run-time system.

5.3 Behaviour Definition Language Design

We believe that a behaviour definition programming language for an NPC definition system needs to be designed according to the requirements laid out in the previous section if it is to cater for the needs of modern computer game development. For the creation of a language which is easy to understand and easy to employ by users – the people who will write programs in that programming language – a number of additional language design related issues need to be taken into consideration. Foremost of these is the understanding of the intended user base, i.e. the system's target audience. This would be game developers, but not restricted to programmers alone. Another contributing factor is what type of language the BDL is going to be. Strictly speaking a BDL in our definition could be called a scripting language, as its programs are not compiled into native machine code but are executed within a run-time module which is embedded within an otherwise independent game engine. Consequently the BDL should be considered an extension language, combining the flexibility of a production language with the power of a task-dedicated scripting system. The restrictions imposed on users by the structure of such a BDL need to be reduced to a minimum and must not interfere with the user's task – instead they need to be harnessed in a way that can empower the user. An example of how this could be achieved is the use of

strongly typed data combined with a reduction of possible data types. While this would slightly reduce the choice available to the user, it would also eliminate possible sources for errors and mistakes.

5.3.1 Design Principles

A BDL that is supposed to be used by non-programmers as well as by programmers needs to be designed accordingly: It is likely that for some game designers the BDL will be the first programming language that they encounter so it is only logical that it should embrace some of the methods used for introductory programming languages. In the context of those requirements McIver and Conway [1996] have identified seven "deadly sins" and design principles and their potential problems and benefits. They argue that a language which has too many different features ("more is more") or too few features ("less is more") or which contains too many syntactical "false friends" ("grammatical traps", "violation of expectation", "excessive cleverness") would make it very hard for users with little programming experience to comprehend the language and to understand what a program does. For the same reason they consider "backwards compatibility" to a similar existing language a hindrance as the prior knowledge of the previous language would only benefit those who already know how to program. Programming languages that are supposed to be used by novice programmers need to have a WYSIWYG[1] character with program source code being able to deliver expected results. McIver and Conway conclude that the ideas they present can only be taken as a guide – not a general solution – and that ultimately the success of the language design can only be measured through user feedback.

Stroustrup [1991] lists five principles that apply to the design of any programming language, and that consequently also apply to the design of a behaviour definition language. These principles can best be described as:

1. Consistency, i.e. the clean integration of features.

[1]WYSIWYG stands for "what you see is what you get", a computer aided design (CAD) paradigm which implies that the output received from the design application will be identical to the final result. In the context of programming languages it is used in terms of predictability, describing syntactic features that closely map to the results of their execution.

2. Modularity, i.e. the possibility of combining existing features to achieve new functionality.

3. Simplicity, i.e. the omission of features for special cases.

4. Performance-neutrality, i.e. the omission of a language feature in a program should not affect the performance of said program.

5. Logical disjunction of features, i.e. the language should allow the existence of programs that do not employ all of the language's features.

A BDL should facilitate object oriented-programming [Stroustrup 1991], as "this paradigm closely reflects the structure of systems 'in the real world'" [Wirth 2006], but, as Wirth notes, it does so as an extension to the traditional programming techniques found in structured programming. Object orientation needs to be regarded as a double edged sword, however, as the additional complexity the object oriented paradigm presents may be overwhelming for novice programmers [Beaubouef and Mason 2005]. Some of the features and mechanisms of modern object oriented languages such as multiple inheritance, polymorphism and exception handling should be avoided or possibly hidden from novice users within a separate access layer to the BDL. This is because they are often confusing for novice programmers as from their "point of view it is simply a case of gratuitous complexity" [Warren 2001].

5.3.2 Resemblance to Natural Languages

We have mentioned before (see Section 5.2.1) that there are arguments in favour of the resemblance of a programming language to a natural language as this may help non-programmers to understand it and use it. Attempts have been made to make existing languages more similar to natural languages by adding various qualifiers and modifiers to keywords and identifiers [Herriot 1977]. Herriot argues that the replacement of abstract structures in programming languages by – among others – adjectives and prepositions to more closely resemble the English language would allow "the program to be its own comments". Some of the presented concepts such as the use of contextual modifiers to allow instances to use the

same identifier as the type definition may be worth further consideration for enhancing the readability of programs. However, most of the changes to the structure of programming languages proposed by Herriot would be mainly of a cosmetic nature and while they would make programs easier to read, they would also make programs much harder to write. Using keywords that result in semantic changes in certain usage situations would make the construction of a compiler for that language much more complicated as it would have to compile context sensitive programs. This is because natural languages are context sensitive and contain too many ambiguities which require additional specification to clarify problems and to resolve these ambiguities. We think that the additional effort required to do this would negate all the benefits gained from the use of a natural language structure in the first place.

The addition of more keywords would also make the use of the language much more error prone. Moreover, linking a programming language's structure intrinsically to a specific natural language would make it much more difficult for non-native speakers of the natural language to write meaningful computer programs, while it would become practically impossible for programmers who do not know the natural language to write programs at all. Providing multi-language versions of a programming language is undesirable, as the language would have to be modified according to the structure of each of the supported natural languages.

Consequently, while natural languages as such may be easy to learn, we believe that their usage would not only make it quite hard to effectively use the language to define NPC behaviour but it would also greatly complicate the overall structure of the system. As a result the computational cost could easily become too large to make this feasible for real-time computer games.

5.3.3 Resemblance to Educational Programming Languages

Some of the most successful introductory programming languages used for the teaching of computer programming employ the "Karel the Robot" paradigm [Anderson and McLoughlin 2006] which relies on the use of a mini-language that provides a small number of instructions and which allow users to take control of

virtual entities, acting within a micro world. The aim for all of these languages is to motivate students to take up programming and to provide them with an enjoyable experience at the same time. The "Karel the Robot" paradigm is named after the very successful "Karel the Robot" program [Pattis 1981], which is one of the widest known computer science teaching tools and has had considerable success. Untch [1990] describes Karel as "essentially a programmable cursor that can move across the flat world" of a 2D grid with obstacles (walls) that cannot be passed and objects (beepers) that can be placed in or removed from the micro world, providing a game-like setting for the task of computer programming.

The use of computer games as the environment for mini-language programmed virtual entities is not a new idea. Apart from the purely educational systems such as "Karel the Robot" there are several examples of games that provide interaction through this paradigm – most of which are available on-line (on the World Wide Web), such as Robocode [Li 2002] or the full 3D action game GUN-TACTYX [Boselli 2004]. In these games the human player interaction is limited to the programming of the entities that "play" the games. The similarity between the control languages in these "programming games" and educational mini-languages clearly shows the correlation of the mini-languages to BDLs for NPCs in computer games [Brom et al. 2006]. The instructions found in the games' control languages as well as in the educational mini-languages are usually a set of actions to be taken by the virtual entities – effectively NPCs – in the virtual environment, as well as a set of (sensor) queries, providing information about the immediate surroundings of the virtual entities in the micro world they inhabit. This micro world provides a graphical representation of the algorithms used in the programs controlling the virtual entities and their position and orientation within the virtual world visualise the current state of the program. This is especially useful for the educational mini-languages as many problems faced by novice programmers can possibly be traced back to an inadequate understanding of program state [Dann et al. 2000]. Among the educational programming languages that use this method of program state visualisation we can usually distinguish between languages that are specially developed to be a teaching tool rather than a language applicable to solving practical problems – a design decision which is often reflected in the choice of an uncommon but possibly more intuitive syntax – and languages

which are directly based on existing production programming languages, providing a more or less complete subset of the "parent" language's syntax, such as our own C-Sheep language with its virtual world of "The Meadow" [Anderson and McLoughlin 2006].

5.3.4 Resemblance to Production Programming Languages

Closely related to these educational programming languages is the Pascal programming language which was meant to be both applicable to real world programming problems but also suitable for the teaching of computer science and programming [Wirth 1993]. For this reason it does not come as a great surprise that many educational "toy-languages", "Karel the Robot" for example (see Section 5.3.3), are based on the syntax of the Pascal programming language. A BDL could therefore be based on a mini-language related production language such as Pascal or a derivative thereof, as is the case with our own ZBL/0 [Zerbst et al. 2003] behaviour definition language (see Chapter 7, Section 7.2). Another possibility would be to base a BDL on the C/C++ family of programming languages which includes the popular implementation programming languages C, C++, Java and more recently the language C#, as well as the scripting languages Perl and JavaScript. One might argue that this approach would complicate the usage of the language. especially if the behaviour definition system is intended to be easily accessible to programmers and non-programmers alike, but we strongly believe that this can be achieved if the system is based on a language which is similar to C/C++. Evidence for this can be found in the film effects industry where many artists have been using complex scripting systems for many years and recently the GPU developer NVIDIA [Mark et al. 2003] has shown with the Cg shader language that artists and shader writers, who may be non-programmers, can understand and effectively use C like programming languages. A further contributing factor for the consideration of a C/C++ like language is the flexibility provided by languages of the C/C++ family which is a necessary precondition for a successful BDL.

5.3.5 Scripting System Design

As explained above, an important part of the process of designing a programming language for novice programmers is the analysis of how many non-programmers who are exposed to a programming language go about using this language. Poiker [2002] explains how novice programmers write programs employing a mixture of "copy and paste"[2] with "trial and error". This stresses the need for extensive debugging support and good language documentation to help users identify and solve problems with their source code, as well as the necessity of a collection of properly annotated (commented) sample source code which can be used as a template by novice programmers. This comes on top of the language features themselves which should include a case insensitive syntax, with an orthogonal structure and strongly typed data types. Tozour notes in "the perils of AI scripting" [Tozour 2002c], that the scripting language design pitfalls which are most destructive to gameplay are:

- a lack of language maturity, i.e. a design which is untried and untested and may not really be suitable for the task for which it is used
- missing development tools and an unsuitable interface which would complicate system usage, program implementation and debugging
- bad real-time performance of the runtime environment of the scripting system
- predictability of scripted events and behaviour through lack of randomization

If the intended user base for the language is carefully considered and if the language is properly designed, then those pitfalls could be easily avoidable, as Brockington and Darrah [2002] point out. Their experience has clearly shown that every scripting system will be used for purposes unforeseen by the system's designers and users will tend to bend the system close to its breaking point. Such a system therefore has to be as flexible and extensible as possible while at the same time

[2] "Copy and paste" is a programming technique in which users copy existing code which has usually been proven to work to reuse it in other places with minor modifications.

being robust and maintaining run-time stability to avoid the kind of catastrophic failure which could disrupt the game engine beyond the point of recoverability.

Chapter 6

Scripting Languages and Computer Games

The games industry is now actively making computer games extensible by allowing the players, to modify the games according to their needs and likes (see Figure 6.1). The method by which this is most often achieved is by using a scripting language. Many developers use well established existing generic scripting systems or permutations of these systems (modified according to the game's requirements) to add scripting facilities to their game. Other games have proprietary purpose-built scripting languages that are dedicated to a single game or game engine. Examples for these scripting languages are QuakeC [Simpson 2002], found in the game "Quake", UnrealScript [BinSubaih et al. 2007], used in games based on the Unreal engine and Scrit [Bilas 2002], the language used in the game "Dungeon Siege".

6.1 Scripting Languages and Scripting Systems

The Oxford Reference Online defines a scripting language as "a programming language that can be used to write programs to control an application or class of applications, typically interpreted" [OUP 2002]. This is only one of many different definitions for scripting languages and this very broad definition encompasses a vast range of programming languages which is – unfortunately – not very helpful.

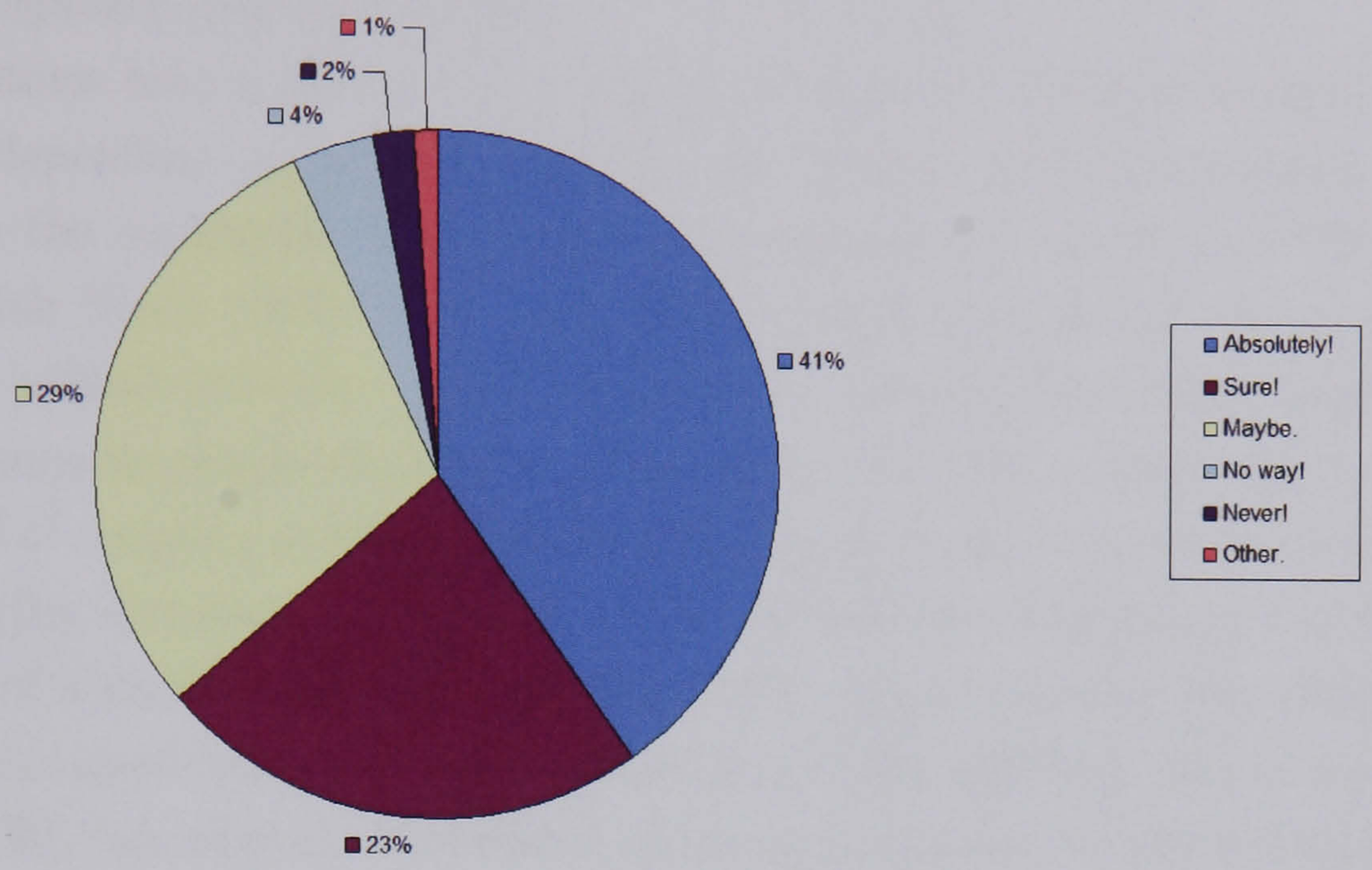


Figure 6.1: Computer game extensibility reasons poll (source: GameAi.com).

When it comes to games, some consider scripting a method for prescribing specific events and behaviour [Sweetser and Wiles 2005], very much like a film script which cannot be altered. We however refer to the terms scripting language and scripting system when we describe a system using a programming language which allows the modification of program logic without the need to recompile the application (game engine) source code.

Scripting languages are used to provide a control interface for combining different components into a single whole, which is why they are also "referred to as glue languages or system integration languages" [Ousterhout 1998]. They are "meant to be easy to program in" [Kerninghan and Van Wyk 1998], often at the expense of run-time performance. As such, scripting languages provide an additional layer of abstraction on top of components (or programs) usually written in a high-level programming language. This abstraction, combined with the fact that modern scripting languages such as Perl [Schwartz 1992] have a lot in common with traditional system programming and implementation languages such

as C and C++, makes scripting languages a form of VHL (Very High Level) programming languages [Bezroukov 2006].

Scripting systems have a wide range of applications and can appear in many different forms. depending on the area of application. Some of the simplest scripting systems are the sophisticated command-line interpreters related to UNIX shells such as Ksh [Korn 1994], their main task being to tie together external programs into a unified construct. Their scope can be greatly enlarged through the use of file processing languages such as AWK [Aho et al. 1979], which form the next higher level of scripting system. Different from these standalone systems are integrated scripting systems such as MEL (Maya Embedded Language) [Gould 2002] that control a single application from the inside, often requiring very little overhead from the application's side for executing scripts, although this is not the case with MEL (see above). Embedded scripting languages are often found in applications for use by non-programmers, i.e. in programming terms "less-skilled personnel" [Wilcox 2007] or "semiprogrammers" [Harmon 2005] for whom programming is not an intrinsic part of their job-description. They include DSLs [West 2007] that can also take the form of macro-based languages that are embedded within an implementation language to be actually translated into native code and linked with its host application [Rabin 2002b], which is a technique considered to be a good use of preprocessor macros [Kernighan and Pike 1999; Rabin 2002a].

While many scripting languages are interpreted, this is not generally the case. Immediate interpretation of scripts which are directly analysed and executed statement by statement is an expensive operation. To achieve a better performance it makes sense to compile script programs, however, not into a frozen executable in native machine code, but rather into an intermediate form for execution within a virtual machine. Scripts that are not interpreted directly but pre-compiled into intermediate interpreter code, running on a virtual machine, can attain considerable performance improvements over those that are interpreted statement by statement, while also preventing some otherwise hard to detect run-time errors by catching them during script compilation. If that compilation happens to be performed on-the-fly, i.e. if the compiler is integrated into the virtual machine as a kind of script preprocessing step, this process is hidden from the

script programmer, providing the illusion that the script is directly interpreted. This is a technique employed by some of the more advanced scripting languages with features that are very close to those of popular implementation languages, showing that they can be a viable alternative to those very same "conventional" programming languages [Prechelt 2003].

6.1.1 A Brief (and incomplete) History of Scripting Languages

Appearing towards the end of the 1960s to early 1970s, the earliest scripting languages were command-based languages that provided more powerful versions of the then common syntax driven user interfaces, allowing operations such as batch processing [Schneider and Nierstrasz 1999]. They allowed for a much more efficient use of the then available file processing filter programs that were capable of interpreting regular expressions, themselves simple languages. The expressiveness of these command-line interpreters was greatly extended with the creation of UNIX shells and the introduction of the pipe which presented a simple method for combining several filter programs [Korn 1994]. This truly showed scripting systems to be an alternative to implementation programming languages, as the combination of existing programs into a different application through scripting allowed the use of a higher level of abstraction, greatly reducing the effort required for solving complex problems [Schaffer and Wolf 1991]. This recognition of the usefulness of scripting led to developments to programs such as the pattern-action language Awk in the late 1970s.

The mid-1980s saw the development of Perl, a language designed to unite the functionality of Awk and the UNIX shell within a single program.

Among the most popular scripting systems in the early to mid-1990s apart from the shell were the languages Awk, Perl and TCL [Kerninghan and Van Wyk 1998; Prechelt 2003], TCL being one of the first embedded scripting languages [Korn 1994] that did not work as an independent command interpreter but had to be integrated with a host application. The need for ever more powerful scripting systems at about the same time led to the creation of systems such as Python, then Lua, soon joined by JavaScript, a development of the emergence of the

world-wide web (WWW), and then the language Ruby, the latter being one of the few programming languages developed in the far east (Japan) [Ierusalemschy et al. 2007].

The late 1990s and the 2000s have seen the rise of the generic embedded scripting language, the more successful of which often have a small memory footprint. At the forefront of this trend resides the scripting language Lua (see Section 6.2.1), leading some to refer to the 2000s as "the decade of Lua" [Harmon 2005].

6.1.2 Comparative Analysis and Classification of Scripting Systems in Games

Just as the term "scripting" has different interpretations, there are different types of scripting systems, each working differently and not all of them are suitable for use in computer games. Our classification of the various types of scripting systems is restricted to those found in modern computer games and does not attempt to be complete but rather means to serve as a guide for distinguishing between different script types. The various types of scripting systems in games are:

ST1 – Initialisation Scripts:

*ST*1 initialisation scripts are the simplest form of scripting system [Tapper 2003]. During program runtime scripts of this type are usually only executed once, at program start-up, while the application is initialising. In most cases this type of script is used only to set internal program parameters to the values given in the script which is why they are also known as "property scripts" [Sherrod 2007]. This is the way we have used this type of script to initialise the application in our evaluation of genetic programming generated computer game players [Anderson 2002]. Initialisation scripts are often nothing more but lists of value declarations, usually interpreted directly and sometimes using additional syntactic elements to make scripts easier to read and edit. This semi-declarative behaviour places initialisation scripts among the DSL family of small programming languages [van Deursen et al. 2000].

ST2 – Trigger-Only Induced Scripts:

In event based scripting systems the occurrence of an event within the game triggers the execution of a script or part of a script. This means that scripts do not

run in a pre-defined order but rather when a specific situation in the game-world has occurred. This category of scripting systems also includes rule-based scripting systems which can be used for the definition of domain knowledge in expert systems, an example of which are intelligent NPCs in many computer games. Commercial computer games that use this kind of scripting system are Bioware's Role-Playing Games "Neverwinter Nights" and "Baldur's Gate". Among the event based scripting systems there are two sub-types:

ST2a – Event Handler Scripts:

The simpler sub-type of scripting systems in this category uses events that are built into its host game engine as predefined events. Here scripts only define the event handlers and possibly additional conditions that may influence the trigger mechanism. Events are triggered and event handlers are called from the game engine itself, when the events occur.

ST2b – Event Oriented Scripts:

The second sub-type are more sophisticated scripting systems that follow the concept of "Action Languages" as described by Gelfond and Lifschitz [1998] (see Chapter 5, Section 5.1.1). Their scripts first define the triggers and the situations in which they should act on events in addition to the event handlers themselves. These trigger-definitions will usually be executed during the initialisation of the scripting system so that these events can be generated by the game engine if all necessary preconditions are met. Once per execution cycle of the script, in many games once every frame, the conditions for triggering events will be checked against the current game-state, i.e. the in-game situation, and if these conditions evaluate as true they will induce the execution of the event handler. The examination of the game-state can happen through active polling of event data from the game engine. Alternatively events can be triggered from within other events or posted as messages to the scripting system by the host game engine.

ST3 – Scripts which Run Like a Traditional Computer Program:

Finally there are the scripting systems that are modelled on "traditional" procedural, functional or object oriented programming languages that would immediately appear familiar to most programmers. Here we can identify two sub-types:

ST3a – LOOPING SCRIPTS:

ST3a scripts will be executed repeatedly to (re-)evaluate the current situation within the game, i.e. they will restart execution from the beginning of the script, once the end of the script has been reached. Effectively, scripts of this type are used to describe a single (high-frequency) control loop. This is the type of script that was generated in our evaluation of genetic programming for computer generated game players [Anderson 2002] (see Chapter 7, Section 7.1). If run once only at program start-up, scripts of this type are also suitable for use in similar environments as scripts of type *ST1*.

ST3b – REGULAR SCRIPTS:

Scripts of this type will execute once only, i.e. they will run from start to finish, concurrently with the host application. Consequently any kind of repeating operation to be executed by the scripting system will have to be implemented as a looping operation within the script itself. An example for this is our mini-language like behaviour definition system ZBL/0 [Anderson 2004] which we will refer to later in this thesis (see Chapter 7, Section 7.2).

6.1.3 Improving Game Design Through the Addition of a Scripting System

In game development, scripting languages are used within the games themselves (by embedding them within the game engines) or in the tools used for game development – usually in situations where the use of an implementation language such as C++ would be inappropriate [Campbell 2006]. A fairly recent poll at the game development website www.gamedev.net – the site is frequented by many game development professionals, as well as amateur developers – suggests that nearly 75% of game engines in development include some form of support for game modifications through scripting systems (see Figure 6.2). Robert Huebner's case study of how scripting support was implemented in the FPS game "Jedi Knight: Dark Forces" details the development process of a proprietary language called COG for use by the designers of the game [Huebner 1997]. COG uses a syntax that is loosely based on the syntax of the C programming language [Kerninghan

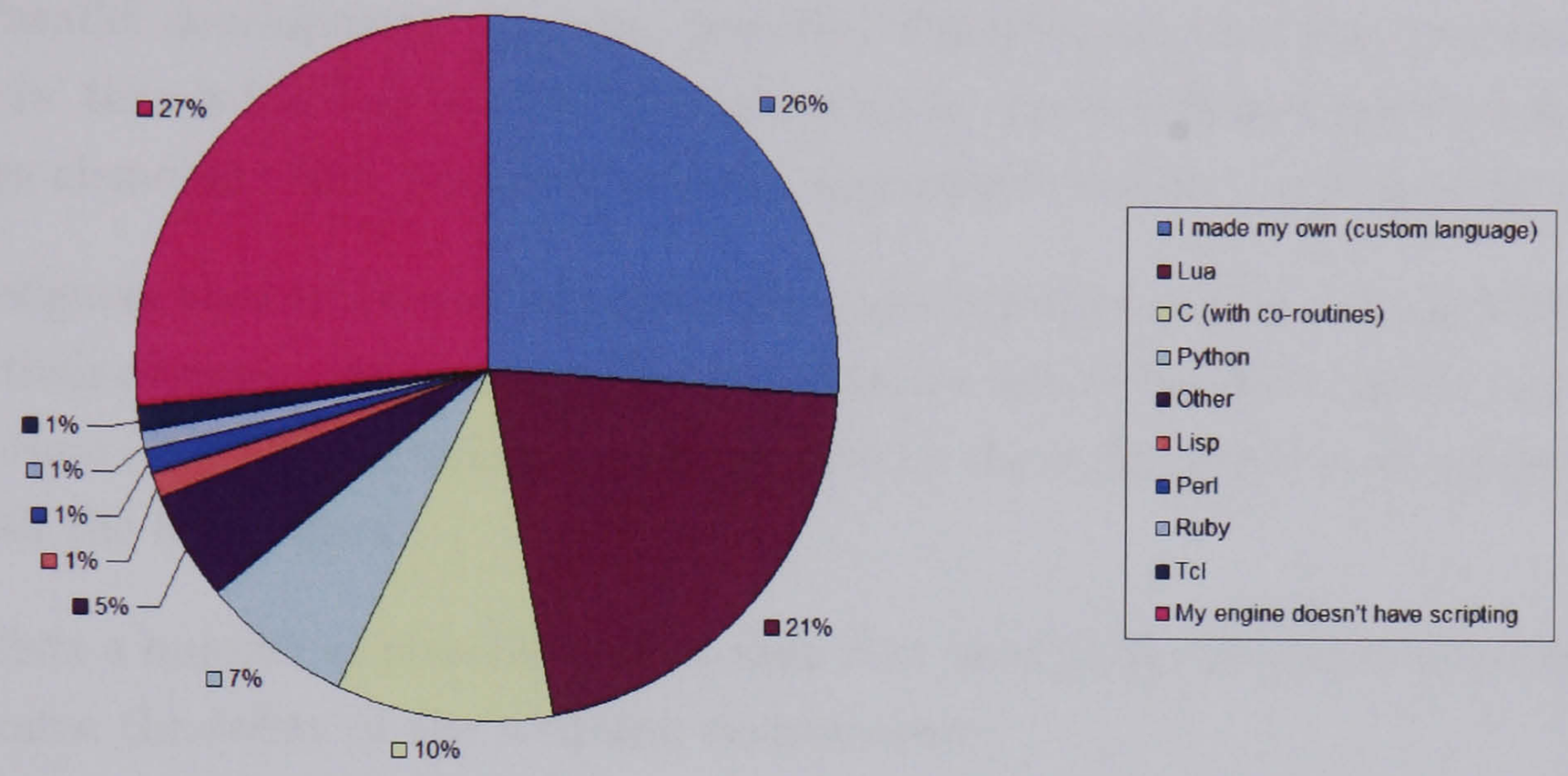


Figure 6.2: Computer game scripting poll (source: GameDev.net).

and Ritchie 1988]. The approach to the design of the language was to use the definition for the C programming language as a starting point and to reduce it until only the desired features were left. Huebner notes that this allowed for a rapid design and implementation of the core scripting system. Similar to C the power of COG does not lie within the core language itself, but within its external functions. These library functions are directly implemented as native functions within the game engine itself and then hooked up to the virtual machine as callback functions. This means that none of the COG library functions executes within the virtual machine of the scripting system but on the computer's CPU, saving a lot of processing time. The virtual machine in the game engine has a stack-based architecture and uses an integrated parser for on-the-fly compilation of COG scripts.

Huebner clearly identifies the benefits of using a scripting system:

- The complexity of the core game engine is reduced as elements of the game logic are taken out of the engine and put into scripts instead.

- The stability of the core game engine is enhanced as a less complex engine design will have fewer vulnerabilities and bugs.

- "Parallel development" becomes possible, which means that the programmers' time is freed up as they no longer need to concern themselves with design elements which designers can now manipulate themselves with scripts.

- Designers are empowered and given the opportunity to realize more aspects of their designs – this is especially true when the virtual machine can do just-in-time (JIT[1]) compilation of scripts and when the script editor is integrated with the level editor.

He also lists a number of possible weak points that need to be taken into account to guarantee the safety of the scripting environment:

- Direct access to game engine variables should be avoided as this could seriously disrupt the engine. A script must not be able to crash the game engine.

- Run-time debugging of scripts must be catered for. If possible source-level debugging should be made available.

Huebner concludes that the design was so successful that designers managed to generate scenarios which would have appeared inconceivable and very hard to realize if it had not been for the COG scripting system. He explains that the similarity of COG to the programming language C not only simplified the development of the language but it also made it easier to learn and understand for the designers – non-programmers – who used COG for the creation of the game. A wide range of documentation and introductory tutorials for the programming language C are available from many on-line and off-line sources and as experience from numerous productions suggests non-programmers can easily be expected to understand and to learn how to effectively use C-like programming languages. This has had a significant impact on the structure of the scripting systems used

[1]JIT or Just-In-Time compilation is an interpretation method in which program source-code is first compiled and then immediately afterwards executed by an interpreter or a virtual machine.

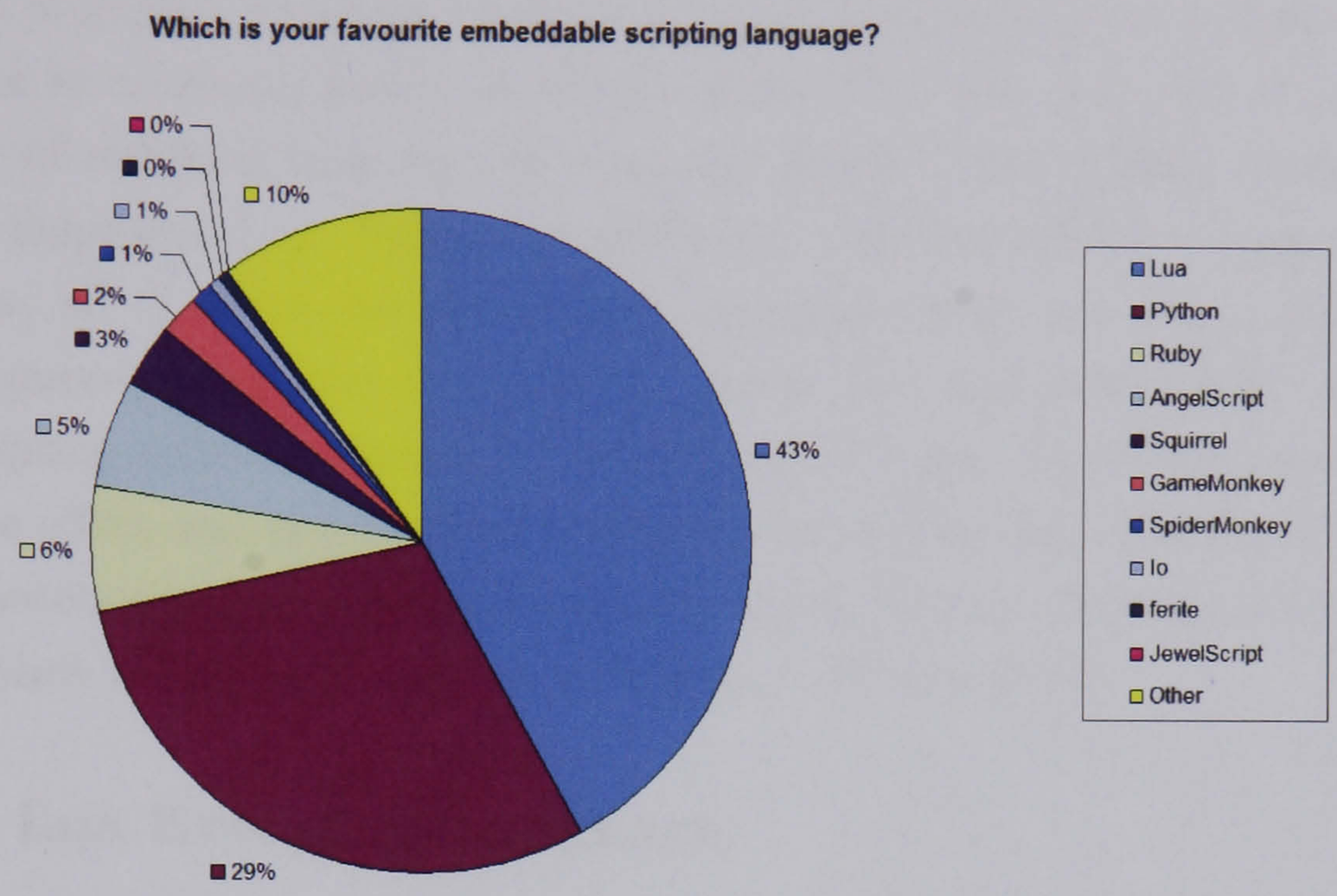


Figure 6.3: Embeddable scripting language poll (source: GameDev.net).

by game developers. Although scripting has been used in game development for quite a long time [Given 2002], access to those scripts has usually been limited to the game developers, and only in recent years the power to modify games has been opened up to the end users, i.e. the game players. Whereas originally the scripting systems were only used in-house by a game's programmers and designers who had direct access to the programmers in case of any difficulties with the system arose, now they have to be developed to a point where they could potentially be "let loose" on the general public where mainly non-programmers would use them to modify the game.

6.2 Frequently Used Scripting Languages in Games

Whereas only a few years ago the majority of scripting solutions used in computer games were proprietary languages (see Figure 6.2), the trend has now shifted

towards the use of generic scripting solutions of which some have been designed explicitly for use in computer games (see Figure 6.3). This becomes evident in a recent survey of scripting languages in computer games [Garcés 2006] which focuses on the languages Lua, Python, AngelScript and GameMonkey Script, presented below, all of which are embeddable languages that have been used in commercial games. This does not, however, mean that the development of proprietary scripting solutions should be avoided at all costs, as Wilcox notes that despite the effort and development overhead involved in the creation of a new scripting system "you are not reinventing the wheel. You are creating a way to concisely express your thoughts in a new language" [Wilcox 2007].

6.2.1 The Lua Extension Language

The scripting language Lua is currently the language of choice for building the scripting solutions in many computer games (see Figure 6.3). It is a generic programming language that was originally designed to be used to extend programs by adding various scriptable features, which is why the creators of Lua have dubbed it an "extensible extension language" [Ierusalemschy et al. 1996]. Lua has a C API, making it easy to embed in C/C++ based applications and Lua is also easy to learn which makes it ideal for game development environments in which non-programmers may be required to write some scripts [Harmon 2005]. The Lua development environment consists of a compiler which can create bytecode as well as an interactive interpreter which allows execution of singular Lua statements. The latter is especially useful for script development as it is a means of generating immediate feedback to a Lua statement which can be tested without full scripts having to be written and compiled.

The Lua run-time environment is embeddable into applications as a portable C library. This library contains a virtual machine, as well as a version of the Lua compiler, allowing on-the-fly compilation of Lua scripts that can then immediately be interpreted by the application into which Lua is embedded in without the need of the scripts to exist in pre-compiled form. If scripts that have been pre-compiled into bytecode are used instead, the run-time environment can be

embedded without the compiler component, reducing its already very small footprint. As of Lua version 5.0 a register-based architecture is used for the virtual machine of the Lua run-time environment, which had originally started as a stack-based abstract machine. This is used to improve program performance, as well as compile-time program optimisation, making Lua one of the few register-based virtual machines used in scripting systems [Ierusalemschy et al. 2005].

Lua is a procedural language which has borrowed features and syntactical elements from a number of existing programming languages. Syntactical influences stem from main-stream languages like C/C++ and there are syntactical similarities to elements of the Pascal based programming language Modula. Another language that influenced Lua not syntactically, but semantically, is the functional programming language Scheme [Ierusalemschy et al. 2007]. One of the more interesting features of Lua is the ability of functions to return multiple values which allows for the creation of powerful scripts for complex situations. Lua uses dynamic typing, i.e. there is no strong typing of variables and only individual values have a data type. Apart from strings there is only one numeric data type which can take floating point values as well as integer values which greatly simplifies the language. The most powerful and useful aspect of Lua however is the use of tables, a dynamic form of associative array inspired by AWK [Aho et al. 1979] and Perl [Schwartz 1992], however implemented in a different, less restrictive manner [Ierusalemschy et al. 2007]. These tables, while very useful by themselves, can also form the basis for much more complex compound data types and even allow the emulation of object orientation.

The language features provided by Lua can simplify the creation of solutions to various problems in the development of computer games, which is why it does not come as a surprise that since its first conception Lua has been used extensively in computer games development, being embedded in a large number of best-selling computer games.

One of the first commercial game developers to adopt Lua were Lucas Arts, a pioneer of the use of scripting in games [Huebner 1997; Given 2002], who used Lua as the scripting language of their GRIME system for the game "Grim Fandango" [Mogilefsky 1999]. Other early adopters of Lua in game development are the company Bioware who used Lua in their action adventure game "MDK2"

[Brockington and Darrah 2002], and Relic Entertainment, who make extensive use of Lua scripting in their games, using a Lua-based system dubbed SCAR (Scripting at Relic) [Rel 2003].

6.2.2 AngelScript

AngelScript or the "Angel Code Scripting Library" is an embedded scripting language, designed with graphics applications and computer games in mind. Like Lua, AngelScript is an extension language, but whereas Lua was designed as an extension language for the C-subset of the C++ language ("clean C" [Ierusalemschy et al. 2007]), i.e. C and C++, AngelScript was designed mainly for embedding in C++, although it has separate C bindings. AngelScript is object oriented but in its current implementation does not yet allow inheritance, although this can be emulated with the system [Shay 2004]. The major differences to similar scripting systems are type safety, i.e. variables that are strongly typed, and the use of native C++ calling conventions for functions in AngelScript which simplifies the integration of scripts with C++ programs, as proxy (wrapper) functions are not required [Garcés 2006].

6.2.3 GameMonkey Script

Another scripting language with C-like syntax, developed specifically for computer games, is the language GameMonkey script [Sherrod 2007]. Unlike the OpenSource AngelScript library, GameMonkey Script started life as a proprietary closed-source language which was later open-sourced. GameMonkey script was created for use with C++, but does not provide an object oriented language itself. In a similar manner to Lua, however, tables can be used to emulate object oriented functionality [Garcés 2006].

6.2.4 Python

Python is a powerful and feature rich scripting language that also allows some object orientation. It is a general purpose language that can be used as a standalone command interpreter but it has also been used as an embedded scripting

environment for various computer graphics applications and also a number of games and game engines [Dawson 2002]. While the language syntax may be considered unusual as it only allows grouping of command sequences into blocks through code indentation, it has an easy to use API which simplifies the embedding of Python into applications, although this is not as easy to do as with the other embedded languages mentioned here [Garcés 2006].

6.2.5 Other Scripting Systems Based on Generic Languages

Other than these popular choices for scripting languages in games, there exist a number of less frequently used but mature scripting languages which can be embedded in computer game engines. Most of these languages are generic, i.e. not specialised for specific tasks [Varanese 2003]. Generic languages of this type that are frequently mentioned in the context of game development are the languages Tcl and Ruby. Other languages used with games are the object oriented language Squirrel or the language JavaScript (standardised as ECMAscript[2] – ECMA-262) which has its origins in web-browsers but has since found a wide range of applications (often embedded through the SpiderMonkey system). An example for this use of ECMAscript is the ActionScript language, which is used in the scripting system found in Adobe's Flash multimedia authoring system, which can also be used for game development [Baba et al. 2007].

The Tcl/Tk scripting system is one of the oldest embeddable scripting systems. The "Tool Command Language" at its core is possibly one of the easiest to learn scripting languages. One of its greatest strengths is the high-level of functionality provided by the Tcl API for embedding Tcl in applications which is why Tcl has also been used to add scripting to game engines.

Ruby is an interpreted object oriented scripting language which is slowly gaining a following among a number of game development teams. Ruby is a relatively new scripting language but it already has a fairly large user community. The commercial quality OpenSource game engine "Nebula Device" has Ruby support (as well as support for Tcl/Tk, Lua, Python and even Java).

[2] http://www.ecmascript.org/

6.3 Scripting Tools for Game Designers

As more and more developers add scripting systems to their games, the need for tools to aid in the development of these scripts has become apparent. Consequently many game developers have created utility programs to answer this need. These tools range from simple text editors that have been extended to provide syntax highlighting for the scripting language to complex CASE (computer aided software engineering) applications that allow an intuitive design approach to the generation of scripts.

6.3.1 Scripting Tools in Popular Computer Games

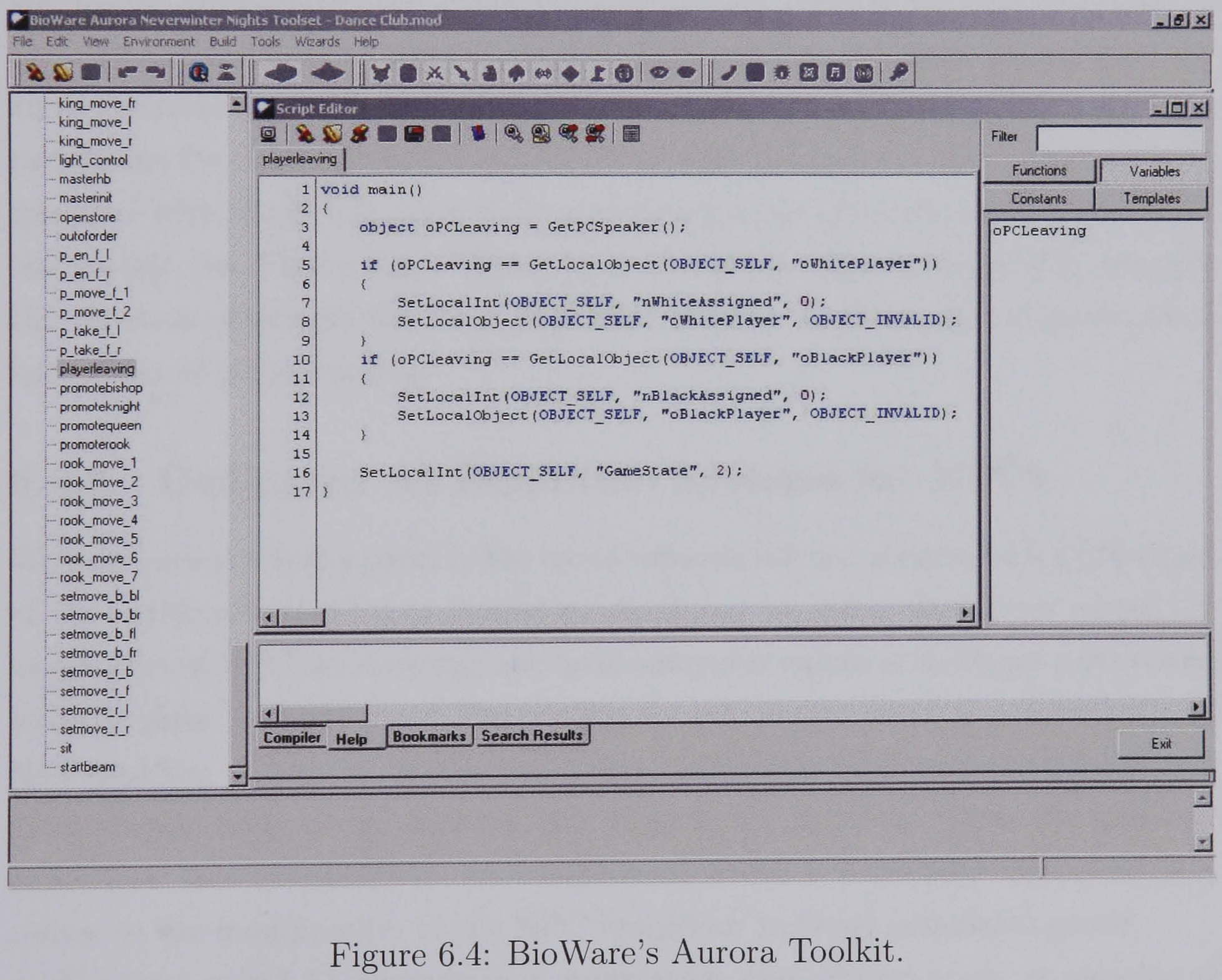


Figure 6.4: BioWare's Aurora Toolkit.

Computer games that can be modified by their user community enjoy great popularity. As a result some of the most advanced script development aids for

extending games can be found in exactly those computer games. These tools are now included in many game releases, effectively making them additional game content [Kane 2007]. One of the largest development communities exists for the Unreal game engine. Not only has this engine been used in a large number of commercially successful games, but the extensibility of these games has resulted in the production of many additional modifications to these games by the players. This has been made possible to a large extent through UnrealEd, the main content generation application for games using the Unreal engine. UnrealEd is not only used for designing the virtual environment of the game worlds but it also contains an IDE (integrated development environment) for UnrealScript, the scripting language used by the Unreal engine [BinSubaih et al. 2007]. A similar tool to UnrealEd is the Aurora Toolkit which is the game editing toolkit of the RPG "Neverwinter Nights". It not only provides methods for building the game environment and placing objects and NPCs for game extensions but also the means for defining the actions of NPCs and the conversations that a player can have with the NPCs using various scripting tools which are embedded within the toolkit (see Figure 6.4). While both of these systems considerably simplify the creation of scripts for their respective games, their use still requires some knowledge of programming.

6.3.2 Dedicated AI Definition Systems for NPCs

We have now reached a point in the trend towards the use of data-driven definition of the artificially intelligent behaviour displayed by game characters where the major part of NPCs in currently available computer games is no longer hard-coded into the game program itself. One reason for this is that developers have realized that enabling players to modify the games themselves adds value to a game and dramatically adds to its shelf-life (see Figures 6.1 and 6.5). Now the question arises how this extensibility can be achieved which is especially important if it comes to the modification of the NPC behaviour in those extensible games.

If a hard-coded AI description is undesirable, one solution to the generation of NPC behaviour from a data-driven behaviour definition is the creation of project-specific proprietary software tools that provide features such as automated FSM

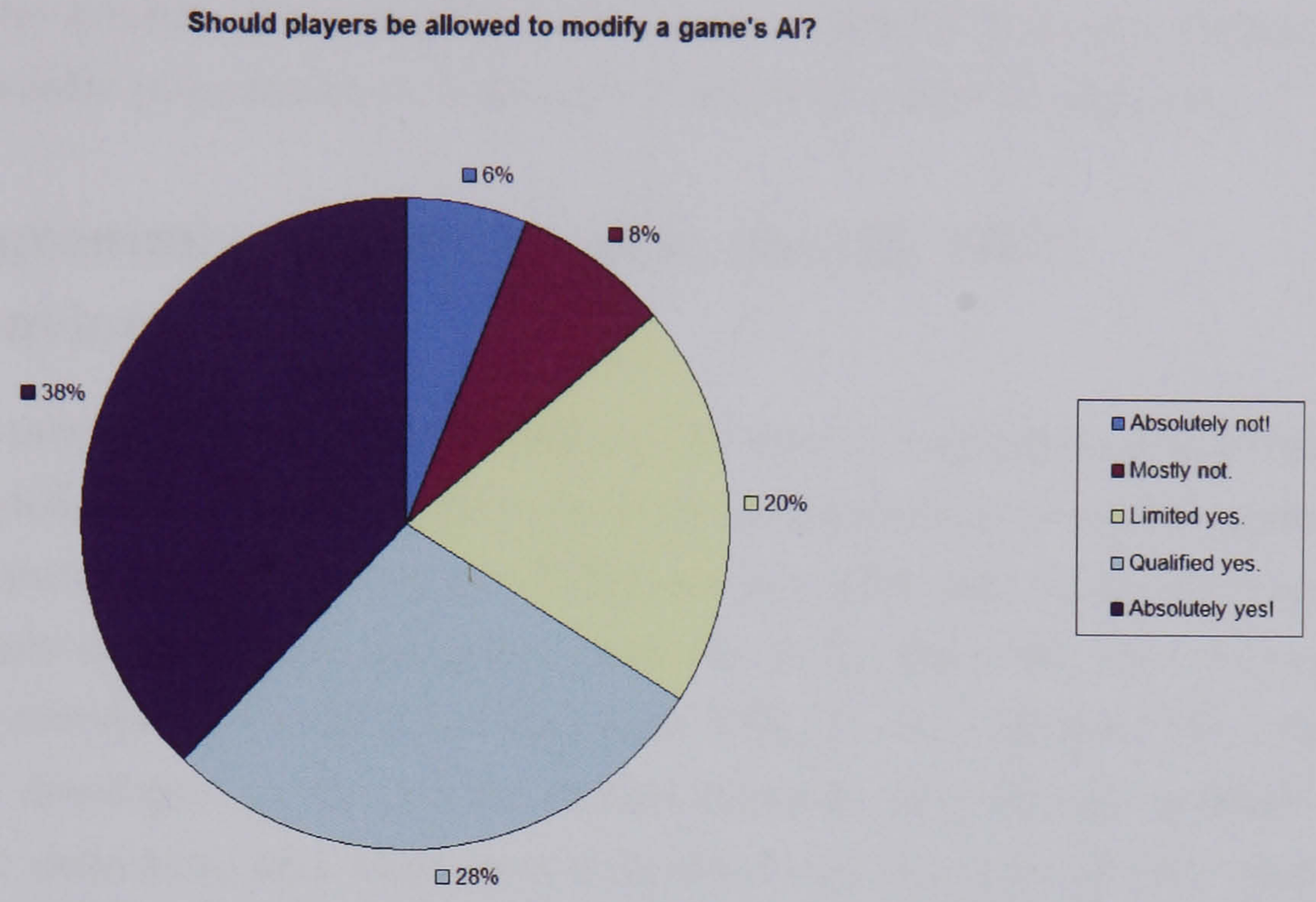


Figure 6.5: Computer game AI extensibility poll (source: GameAi.com)

(Finite State Machine) generation [Jacobs 2005]. While most of these applications are created in-house, a number of 3rd party developers have attempted to create more generic systems which are not bound to specific projects. Games can settle for these commercially available middleware systems [Dybsand 2003; Kruzewski 2006] or alternatively they can use a scripting language of some sort (established or proprietary). A scripting system might seem an ideal solution for the complexity of the problem but there is not one single method by which the behaviour of artificially intelligent characters is created and therefore a solution found for one game is not easily transferable to other computer game productions. This is especially true when it comes to the scripting of NPC behaviour as some of the different kinds of scripting systems which are used in conjunction with AI in games are quite generic and are not exclusively used for scripting the AI, but also for other tasks within the game.

There are some dedicated AI scripting systems that have been used in a number of games and animation systems. In most cases they have been highly specialised for specific genres of computer games or kinds of behaviour that is

generated by the system. This greatly restricts the reusability of such systems and they are usually proprietary to a specific product or range of products.

6.3.3 Programming Solutions that Modify NPC Behaviour

The design of a proprietary behaviour definition (scripting) language and run-time system for the definition of artificial behaviour as an extension to a specific game or genre of computer games (for example FPS games) is relatively straightforward and simple if only deterministic behaviour is involved. To illustrate this one can take the first prototype for the ZFX bot language (ZBL/0 – see Chapter 7, Section 7.2) which was developed by the author of this thesis to demonstrate syntactic NPC behaviour definition, and which was completed over a period of little more than a fortnight (from conception to first use) [Spirig et al. 2003].

In effect such a language can take the form of a DSL which needs to do little more than provide a function binding interface to a game engine, allowing the creation of simple rule based systems. In situations like this the game engine itself does all the work while the script only ties together the different game engine components that provide the NPCs with functionality. This is especially true in simple cases where the sole use of scripts is the initial configuration of otherwise hard-coded NPC behaviours using initialisation scripts of type $ST1$ (see Section 6.1.2) [Tapper 2003]. The most complex scripting solutions are programs that use high-level abstract descriptions to define complex behaviours. Languages of types $ST2$ or $ST3$ are a lot better suited to the definition of complex behaviours than scripting languages of type $ST1$. The development of this type of system from scratch can take considerably longer, so a good solution might be to base this behaviour definition language on an existing AI language (see Chapter 5, Section 5.1.1) or to use a generic embeddable scripting language.

6.3.4 Visual Script and NPC Generation

Although non-programmers can cope perfectly with writing programs in a scripting language, one approach to simplifying script generation for designers is the

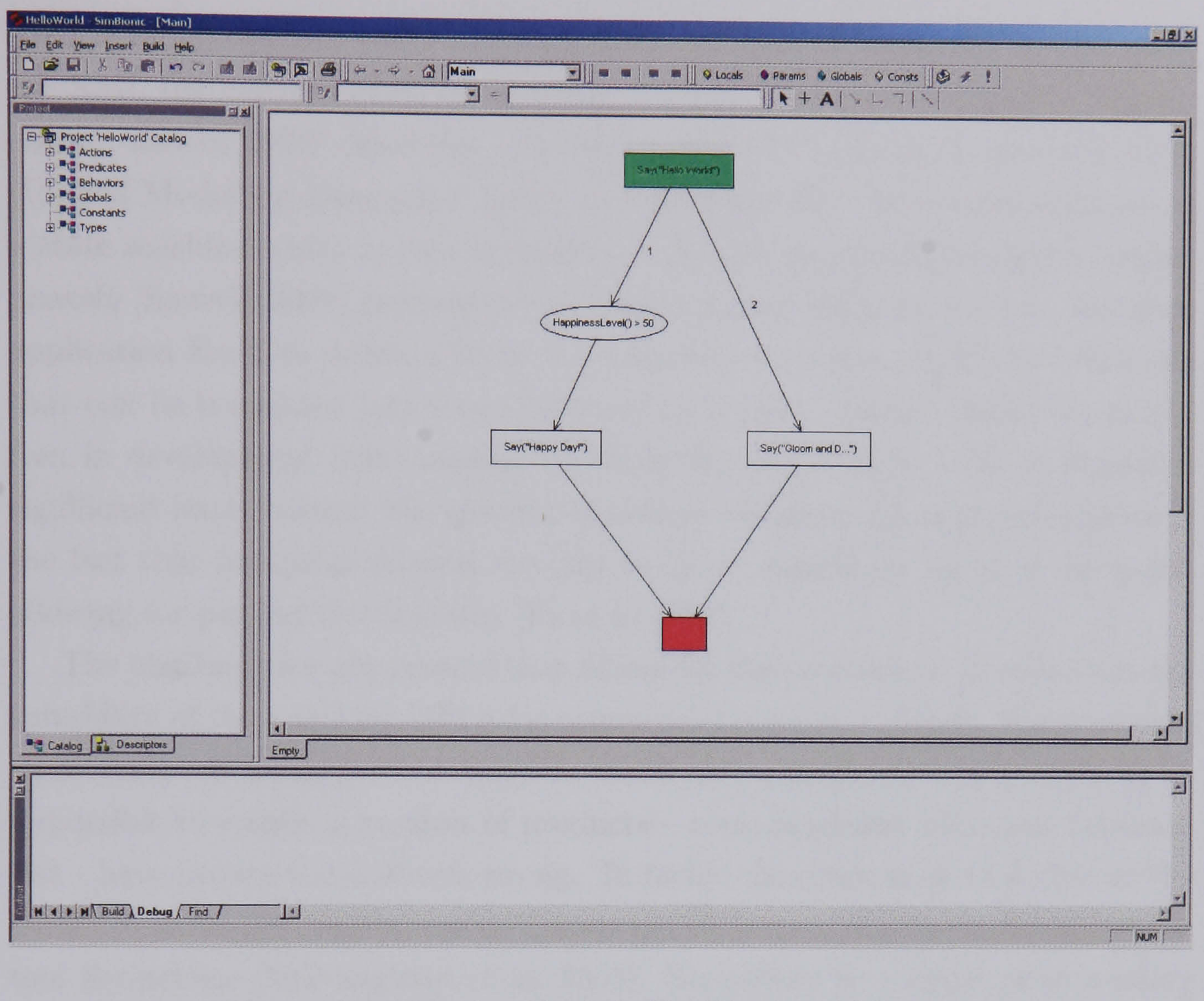


Figure 6.6: Stottler Henke's SimBionic middleware.

use of tools with a graphical user interface (GUI) that provides designers with an intuitive tool for designing NPC behaviour [Snavely 2006] and presents a visual representation of the script that will be generated [Houlette et al. 2001]. A WYSIWYG paradigm is especially useful for taking a complex behaviour description in natural language, such as one found in a design document, and transferring it into a form that can be visualised and interpreted by a run-time system for controlling NPCs [Houlette et al. 2003]. Through the provision of a small set of hierarchically connectable graphical elements the rapid construction of expert systems for this task by an expert of the domain who needs no understanding of programming becomes possible. Carlisle [2002] proposed the implementation of a GUI-based design tool for the definition of state machines. Gill [2004] uses the

diagramming software Visio to design state machines, storing the results using the XML (extensible) mark-up language which is then used as input in a game engine. Jacobs [2005] describes a similar system which allows the use of a UML (Unified Modelling Language) design tool to create the visual representation of a state machine which is then translated into code useable in the game engine. Snavely [Snavely 2004] proposes to use a spreadsheet program, such as the office application Excel to define a fuzzy state machine to create intelligent behaviour that can be translated into a useable form for a game engine. While the reduction in development time achieved through the use of these tools is already a significant improvement, the greatest benefit to the game development process is the fact that non-programmers are able to create significant parts of the game, allowing for parallel development [Fu et al. 2003].

The challenge for any system that allows for the creation of AI entities is the suitability of the resulting NPCs for commercial computer games. Whereas only a few years ago many game developers would have considered such systems to be impossible to create, a number of products – some academic and some commercial – have proven the doubters wrong. To tackle the creation of AI scripts in the RPG "Neverwinter Nights" researchers at the University of Alberta developed the tool ScriptEase [McNaughton et al. 2003]. ScriptEase is a visual programming application that specializes in generating scripts for RPGs. Unlike the more comfortable to use "point&click"[3] graphical front-ends that generate the NPC code from flow diagrams, ScriptEase provides a menu and dialog-based user interface for the definition of NPC behaviours. A similar dialog-based scripting method using context-sensitive menus is used in the educational system Alice [Kelleher 2006] to simplify script programming. A different approach to the development of AI characters for "Neverwinter Nights" was taken by the developers of the freely available BrainFrameNWN Editor, which is a specialised version of the visual authoring tool used in the commercial AI middleware system SimBionic (see Figure 6.6). The BrainFrame Editor is used to graphically create flow diagrams of a hierarchical representation of NPC behaviours [Fu and Houlette 2002]. This

[3] "Point&click" describes a type of user interface which is mouse-driven, i.e. the mouse pointer is used to select objects, while clicking the mouse buttons manipulates the selected objects.

representation is then used to generate the necessary data for the run-time system used by the game engine or in the case of the BrainFrameNWN for the code generation in NWScript. Finally, the concepts of iconic programming [Calloni and Bagert 1995] need to be mentioned as they could be used to create a hybrid representation of the two approaches described above, combining a visual control flow representation with a graphical representation of programming language instructions.

6.4 Systems for Syntactic Behaviour Definition

We are not aware of any existing single, common method for the implementation of a game character AI. Existing solutions usually require the combined use of several techniques. A better solution would instead provide a smooth integration of the behaviour definition system into game applications within a unified architecture that provides a combination of all of the different components for creating NPCs. Further improvements in the quality of computer games are likely to have to adopt a syntactic approach for the creation of the artificially intelligent characters that populate the virtual worlds of computer games. This is likely to take the form of a high-level programmable system for defining NPC behaviours. We have defined the term "Behaviour Definition Language" (BDL) to describe such a high-level programming language and we have set out the criteria that need to be met by a syntactic behaviour definition system to be a BDL. We have also described several systems for NPC behaviour definition, many of which fulfil several of the criteria for BDLs, but to our knowledge so far there exists no system that fulfils all of them. The first step towards a unified software package for creating life-like NPCs in computer games is the creation of a BDL as a system for syntactic behaviour definition, the development of which is the subject of the following chapters of this thesis.

Chapter 7

The Development of Three Behaviour Definition Languages

The previous chapter provided an overview of scripting systems and scripting languages with a special focus on the use of generic embeddable scripting languages in computer games. In this chapter we discuss three behaviour definition languages that we have created in the course of our work.

7.1 GP Asteroids Script

GP Asteroids Script is the result of a series of experiments for using evolutionary computing techniques for evolving an artificial player, capable of competing in the popular arcade game "Asteroids"[1]. A detailed description of our experiments

[1]The classic arcade game Asteroids is based on attack and evasion which is a concept that is common to most action oriented video games. In Asteroids the player's spacecraft flies through a two-dimensional field of moving asteroids. The player has to avoid colliding with the "asteroids" to prevent his space ship's destruction. At the same time he has to destroy the asteroids to win and progress to the next level of the game. To achieve this, the player can shoot at asteroids. If hit, a large asteroid will break up into two medium sized asteroids which in turn can each be split into two small asteroids. Shooting down an asteroid increases the player's score. The player has a pre-defined number of lives. A collision with an asteroid destroys his spacecraft. In our implementation of the game the players only means of defence is to use a protective shield, which protects the player from destruction during collisions with asteroids by granting temporary invulnerability while it is active. Using shields or firing the gun will drain the player's energy which is replenished over time. The game ends when the

and results using this system is presented in [Anderson 2002]. In our version of Asteroids the player indirectly interfaces with the game through a Lisp [McCarthy 1959] like behaviour definition (scripting) language which implements a number of sensors and controls, which are identical to a human player's controls if he were directly interacting with the space ship (see Appendix B for a description of the language). The script program which controls the player entity is created through off-line evolution, based on the agent's proficiency at playing the game. The space ship's controls are used as output of the evolved program, whereas the space ship's sensors, which reflect the current state of the game, are used as input to the evolved program. GP Asteroids Script is a scripting language of the Looping Script type ($ST3a$ – see Chapter 6, Section 6.1.2), i.e. the complete script is evaluated as a whole for every update cycle of the game, i.e. once every frame.

The run-time system used to evolve GP Asteroids Script programs, as well as execute them, is a version of Sarafopoulos's "gp interface" library [Sarafopoulos 2001], extended with the player specific instructions for controlling the space ship. Of these instructions most are used to set and query the player entity's states (in the game) which are implemented through an FSM. Only a few of the instructions can be used to immediately trigger actions for execution during the current update cycle of the game.

7.1.1 The GP Asteroids Script Programming Language

GP Asteroids Script is a small functional programming language with only 23 unique instructions (excluding operators and constants – see Table 7.1). The language was designed for a variation of genetic programming (GP) [Koza 1992] that is "strongly typed", as introduced by Montana [1995], which allows for the use of different data types. The language has two data types, one for Boolean values which can take on the truth values 'TRUE' or 'FALSE', and a typeless data type which is used for procedures that do not return any data. For this three

---

player has lost all of his lives. The aim of the game is to stay alive as long as possible and to gain the highest score during that period.

constants (two Boolean: 'TRUE', 'FALSE' and one typeless: 'void') are defined. which can be used with the language's control structures.

These program flow control structures are implemented as non-terminal functions for:

- a dyadic (if-else) selection
- a comparison (if-equals)
- a sequence of two instructions

This small set of control structures is augmented by four simple Boolean operators (AND, OR, XOR and NOT) which are implemented as non-terminal functions of the language.

The player entity's sensors and controls make up the terminal set of the language's functions. The former, i.e. the sensor queries for the space ship, are implemented as a set of Boolean functions, whereas the latter, i.e. the controls for the space ship, are implemented as a set of action procedures which enable the player entity to switch its current state. The functions available to retrieve the space ship's sensor states return information regarding

- the current level of the space ship's energy
- the state of the ship's movement (forward and turning motion)
- the approximate positions of targets (asteroids) in relation to the player's position

The actions available to the player are

- setting a state for turning (left, right, not)
- setting acceleration (on, off) or deceleration (automatically reset for each frame)
- setting the state of the shields (on, off)
- firing a single bullet

These functions give sufficient control to scripts, allowing the construction of simple player entities that can play the game.

sensor functions	
	(targetAhead)
	(targetLocked)
	(proximityAlert)
	(impactAlert)
	(hasEnergy)
	(plentyEnergy)
	(hasShields)
	(lookingAhead)
	(isMoving)
	(accelerating)
	(isTurning)
action functions	
	(setThrust)
	(noThrust)
	(decelerate)
	(setShields)
	(noShields)
	(rightTurn)
	(leftTurn)
	(noTurn)
	(fire)
control structures	
	(if_true)
	(if_equal)
	(sequence)

Table 7.1: GP Asteroids Script functions.

7.1.1.1 Automatically Defined Functions and Super-Actions

In a separate set of experiments we used a modified version of GP Asteroids Script that allows the use of up to three so-called automatically defined functions (ADFs) with some additional syntactic constraints to the structure of script programs. All of the ADFs are defined as terminal functions that return void and take no parameters, each of which is allowed to contain all of the available functions, procedures and control structures. The constraint to the program structure is that the result-producing branch (RPB) which contains the main program script only allows the use of the control structures (see Table 7.1) and the three ADFs.

This modified version of the GP Asteroids Script language is tailored to fit the requirements of the game at the expense of choice of instructions for the programmer of the player program. The goal of the game Asteroids is to maximise the player's score, usually by destroying all asteroids as quickly as possible, and a precondition for the destruction of all asteroids is the player's survival. This analysis of the problem-space leads to the identification of three distinctive, yet partially conflicting objectives from which separate behaviours can be extrapolated:

- destroying a target which is in the player's range and line of fire – aggressive behaviour
- seeking out and finding targets in the shortest possible time – hunting behaviour
- avoiding collisions with asteroids – defensive/evasive behaviour

This leads to the identification of the three possible ADFs – 'Aggression' for aggressive behaviour, 'Defence' for defensive behaviour and 'Target Acquisition' for hunting behaviour. Each of the three objectives is associated with a different ADF. The use of segregated branches of the parse tree for achieving multiple objectives as described by Reynolds [1994] was the inspiration for the use of ADFs to find a solution that successfully completes the three conflicting objectives of the Asteroids game. To ensure that each of these three ADFs evolves in a way that will satisfy its assigned objective, the fitness evaluation of individual players is distributed using task specific fitness functions during program evolution, i.e.

the GP system uses a separate fitness function for each ADF which evaluates the fitness of that ADF for its assigned task. It was necessary to impose the syntactic constraint of only allowing ADFs and program flow control structures in the RPB, as early experiments showed that allowing the use of all instructions in the RPB can lead to functions and procedures in the RPB cancelling out the results generated by the ADFs.

For testing purposes an additional set of three super-actions, each equivalent to one of the three ADFs ('fireAtWill' for aggressive behaviour, 'seek' for hunting behaviour and 'autoprotect' for defensive behaviour) were added to the instruction set of the language, to be used instead of the ADFs. A successful player using a GP Asteroids Script program written with these super-actions would be the following:

```
(sequence
  fireAtWill
  (sequence
    autoprotect
    seek))
```

This program executes the three possible super-actions in sequence. Each of the actions includes a conditional evaluation of the current game state which determines if the super-action is executed, making the super-actions a kind of state and the whole program some kind of finite state machine description.

7.1.2 Designing Artificial Players Using GP Asteroids Script

To illustrate the usage of GP Asteroids Script the program below shows how a player script would look. The version of the language used here is the one using automatically defined functions (see Section 7.1.1.1 above). This means that from the top-level function, player scripts are only allowed to use the program flow control structures and to call the three automatically defined functions ('ADF_1', 'ADF_2' and 'ADF_3') for aggressive, defensive and hunting behaviour.

The main function for a successful player would call all of the ADFs in sequence:

```
(sequence
  ADF_1
  (sequence
  ADF_2
  ADF_3))
```

Assuming, that the first ADF defines aggressive behaviour (equivalent to the 'fireAtWill' super-action), the controlled entity should fire its weapon if it has sufficient energy to power its gun and if it has a target in its sights. A possible 'ADF_1' description could therefore be written as shown below:

```
(if_true
  (AND
    plentyEnergy
    targetLocked)
    fire
    void)
```

The second ADF (assumed to be equivalent to the 'autoprotect' super-action) should prevent the player entity from being hit by asteroids. This can be achieved by activating the player entity's shields if an asteroid is about to impact with the space ship and deactivating the shields to conserve energy if there is no imminent danger. As an additional defensive measure the player entity should also move away from its current position. The resulting 'ADF_2' would look as follows:

```
(if_true
  impactAlert
(sequence
  setShields
  setThrust)
```

noShields)

The most complicated function (equivalent to the 'seek' super-action) is the third ADF 'ADF_3', displaying hunting behaviour:

```
(if_true
   proximityAlert
   (sequence
      (if_true
         isMoving
         (if_true
            impactAlert
            (sequence
               noTurn
               decelerate)
            (sequence
               decelerate
               noThrust))
         void)
      (sequence
         (if_true
            (AND
               (NOT
                  targetLocked)
               (NOT
                  impactAlert))
            rightTurn
            noTurn)
         (if_true
            impactAlert))
         (if_true
            plentyEnergy
            fire
```

```
            void)
          void)))
    (sequence
      (if_true
        targetLocked
        void
        leftTurn)
      setThrust))
```

The listing above shows a possible method for target acquisition in the hunting behaviour. Here the first action of the player entity is to check if an asteroid is close by ('proximityAlert'). If not, the player entity either accelerates towards a target, if one is in front of it, or it rotates to the left, if there is no target in sight. If an asteroid is close by and the player entity is moving, it stops turning and slows down, if there is a threat of impact or just decelerates if the space ship is not directly threatened. If there is no asteroid in sight at this point, the space ship rotates to the right, however if an asteroid is within the player entity's sights and it has enough energy, it will open fire.

7.1.3 Concluding Remarks on GP Asteroids Script

While the experiments to automatically generate an artificial player capable of playing asteroids were successful [Anderson 2002], the usefulness of this system for creating NPCs as such is questionable. From a game development standpoint, the system is very restrictive, i.e. not easily extensible, as all extensions would have to be added directly to the source code of the system. The absence of a looping control structure works only because the whole of the program is evaluated for every update cycle, but as a behaviour definition language GP Asteroids Script is effectively incomplete. Furthermore, the language's existing instruction set limits the system to use in games of a small sub-genre of arcade action games, namely 2D space shooters with 360 degree of freedom of movement, i.e. games such as Asteroids, "Computer Space" or "Space War".

We do not believe that systems of this type, using a Lisp-based scripting language, scale well. While it is easy to write programs using GP Asteroids Script, they quickly reach a state in which they are too complex to be easily deciphered (see code listings in Section 7.1.2). This is due to the recursive syntax of the Lisp-like language, and while it would be possible to add additional instructions for sensor inputs and actions to make the language more versatile, script programs themselves could easily grow to a point where they are unmaintainable, due to a lack of visually recognisable structure.

7.2 FPS NPC Behaviour Definition Language ZBL/0

The embedded Regular Script type (*ST3b* –see Chapter 6, Section 6.1.2) behaviour definition (scripting) language ZBL/0[2] (effectively pronounced "Sybil-Zero"[3]) is a problem-oriented third generation programming language. ZBL/0 is a simple to learn, simple to understand and simple to embed scripting language for defining the behaviour of NPCs in real-time FPS computer games, which are sometimes also referred to as bots or game-bots. As such, we developed ZBL/0 as the scripting system for the creation of computer controlled opponents for the FPS game Pandora's Legacy [Zerbst et al. 2003]. The ZBL/0 scripting system consists of a compiler for game-bot programs (NPCs) that have been written in the ZBL/0 language and a robust virtual machine that can be integrated into any game engine.

ZBL/0 is much smaller, more restrictive and far less extensible than many other scripting systems, i.e. the language is dedicated to only one genre of computer games and the virtual entities that populate them. Following the example of mini-languages found in computer science education [Brusilovsky et al. 1997], the ZBL/0 language is based on a traditional programming language which has been reduced to the simplest features to make the system easily accessible for

[2]http://zbl0.zfx.info

[3]The acronym ZBL/0 is pronounced using American English pronunciation of the letters Z (zi) B (bi) and L (l) and the number 0 (zero). The resulting word is pronounced similarly to the female first name Sybil.

programmers and non-programmers alike. The syntax of the ZBL/0 language is influenced mainly by the PL/0 model language [Wirth 1986] and therefore bears some resemblance to the high-level procedural programming language Pascal [Wirth 1993] with some additional influences for syntactical constructs that resemble the programming language C [Kerninghan and Ritchie 1988]. Unlike C, however. ZBL/0 is case-insensitive, i.e. the differences between lower-case and capital letters are ignored and all instructions and identifiers can be written in lower-case, upper-case or a combination thereof. Comments used in ZBL/0 are line-comments, similar to those found in C++, rather than the block comments found in languages such as C or Pascal.

7.2.1 The Design and Development of ZBL/0

ZBL/0, the ZFX[4] Bot Language, was developed towards the goal of creating a simple method for adding NPCs for computer games and provides an open standard for that aforementioned purpose. The design of the ZBL/0 language and system was guided by the aim to create a tool for learning how to syntactically define NPC behaviour in FPS (First Person Shooter) games which we used as a reference system to illustrate the development of game-bots in the context of a book on computer game development [Zerbst et al. 2003].

The requirements for the ZBL/0 scripting system were straightforward:

- The system was to be used to define NPC behaviour as an extension to computer games of the FPS genre written in C++.
- The NPCs defined by the language only needed to support deterministic behaviour.
- No complex data types or control structures needed to be implemented as the system was supposed to be used to demonstrate general concepts of NPC behaviour scripting.

[4] "ZFX – 3D Entertainment" (http://www.zfx.info) is a German language on-line community dedicated to the development of computer games.

Consequently the development of the system from conception to first use was achieved in a very short period of time: The first fully working prototype for the ZBL/0 system for example was completed over a period of little more than a fortnight [Anderson 2004].

The starting point for the definition of NPC functionality that the system was supposed to be able to describe were the NPC control functions found in GP Asteroids Script (see above). The choice of functions was furthermore informed by an examination of common player controls in FPS games in order to mirror the functionality exposed to human players in FPS games within the scripting language's set of functions. This instruction set was refined during the development, reflecting discussions with Spirig et al. [2003] (personal communication). To simplify the structure of the ZBL/0 system the instructions for NPCs were added to the language as intrinsic functions. The function identifiers themselves were selected to be self explanatory for easy understanding. The core language was augmented with additional syntactical elements similar to those found in C/C++, such as the use of parentheses to encapsulate conditional expressions in selection statements and loops, to simplify the process of writing ZBL/0 programs for programmers that are experienced in C/C++ [Spirig et al. 2003]. The use of these additional syntactical elements was made optional, i.e. they can be used in programs but they can also be safely omitted from programs.

One of the goals was to find a simple way of embedding the ZBL/0 system into game engines written in the C++ programming language, currently the most common choice of language for computer game development. An important part of the embedding process is the binding of script functionality to functions defined in the host application, i.e. the game engine. If the programming language for the system had been C, the obvious choice for this would have been the use of function-pointers (pointer variables that can hold the address of a function) to create callback functions[5] for the function bindings. The method chosen, however, was to make use of the object oriented features of C++ and to achieve the function

[5]Callback functions are functions which are not explicitly called but instead are invoked implicitly by the program, i.e. control over the sequence of functions called is removed from the control of the user (the programmer). Callback functions are usually used as event handlers that need to be "registered" with the calling application so it knows which callbacks to use.

bindings through polymorphism. Game-bot objects are derived from an abstract base class (through inheritance) which can then be passed to the virtual machine. User errors caused by the omission of methods that need to be implemented are prevented by the rigid structure of this function binding interface which only provides declarations and places the burden of the implementation onto the host application. The virtual machine knows the game-bot functions in the abstract base class which are equivalent to the functions of the ZBL/0 language and can call them if they are invoked by a ZBL/0 program, so all that is required of the programmer is to realise the NPC's functionality in the game engine is to implement the methods in the game-bot class that was derived from the abstract base class described above.

The first step towards the implementation of the ZBL/0 system was the creation of a first detailed specification of the ZBL/0 language which was followed with the creation of a series of prototypes for the language's compiler and virtual machine which then underwent a series of step-wise refinements. ZBL/0 is defined using an LL(1) grammar (see Appendix C). Programs that are compiled with the ZBL/0 toolkit are parsed using a recursive descent parser followed by generation of bytecode for the system's virtual machine. The compiler and the virtual machine were progressively refined until their capabilities and the specification of the language appeared to be suitable for the task at hand, i.e. simple enough to use for novice users, but powerful enough to be deployable in real-world applications.

7.2.2 The ZBL/0 Programming Language

As a behaviour definition language that resembles educational mini-languages, ZBL/0 provides a task specific set of instructions and queries which allow the programmer to take control of virtual entities acting within a micro world. In the case of ZBL/0 the virtual entities are NPCs and the micro worlds they inhabit are the virtual environments of FPS games which is reflected in the functions and procedures of the language (see Section 7.2.1 above).

The current version of the ZBL/0 language only supports a limited set of control structures (simple iteration, condition/alternative and sequence) and the

alive	armour	back	backstep
blocked	crawl	danger	die
duck	face	find	fire
front	health	idle	initialize
jump	jump_back	jump_left	jump_right
jump_up	left	memorize	object
object_ahead	obstacle	owns	respawn
right	rnd	spawn	spawned
step	strafe_left	strafe_right	target
target_ahead	target_alive	target_armour	target_health
turn	turn_left	turn_right	use
using			

Table 7.2: ZBL/0 intrinsic functions.

definition of simple procedures and functions, however in those user-defined functions there is no language support for function parameters. Instead function parameters have to be emulated through the use of global variables in the behaviour definition program. Every user-defined function in ZBL/0 can have local sub-routines, i.e. functions with local scope. There is only one variable data type in ZBL/0 which can be used to store numerical values (integer as well as floating point), which is automatically used as the return type for functions. User-defined functions always implicitly return the value '1', unless a different value is explicitly returned.

The function set for controlling game-bots is intrinsic to the ZBL/0 scripting language, i.e. built into the language (see Table 7.2). As a result the use of functions does not have to be enabled by means of inclusion of a library of functions. This intrinsic function set consists of 45 functions representing actions and sensor queries that can be performed by an NPC in FPS games, such as turning towards an opponent, moving in a specified direction or firing a weapon (see Appendix C for a detailed description). The functions of ZBL/0 can be sorted into three distinct categories:

1. housekeeping (game-bot management) functions

2. sensor query functions

3. action control functions

Of these, the housekeeping functions of ZBL/0 include those that govern the initialisation of game-bots as well as any functions that directly impact the existence of NPCs in the virtual game world, such as the querying of world state information that is not sensor information but might prompt a game-bot to change its behaviour.

The ZBL/0 sensor query functions provide a behaviour definition program with the information an NPC perceives about itself and its environment. This data allows the game-bot to orient itself in that virtual environment.

The final set of ZBL/0 functions includes the action control functions. These include the functions to control a game-bot, allowing it to interact with its environment within the virtual game world.

7.2.3 ZBL/0 Virtual Machine

The ZBL/0 virtual machine is a kind of parallel stack-oriented machine, written in platform independent ANSI C++. It allows the creation of several simultaneously running processes with each game-bot process running in its own, self-contained, micro-thread [Dawson 2001], each of which has its own stack to keep different programs separate. In addition to the stacks themselves the registers for each process (program counter, program instruction register, base address register and stack register) are encapsulated with each stack to provide a separate entity. The virtual machine uses pre-emptive multi-tasking with round-robin[6] scheduling to execute loaded processes during its run cycle. This means that whereas game-bot processes are executed sequentially in the embedded virtual machine, from the outside, i.e. the host application, the virtual machine appears to execute several programs in parallel. The ZBL/0 virtual machine is a self-contained module and

[6]Round-robin is one of the oldest and simplest scheduling algorithms [Tanenbaum 2001], used in many multi-tasking systems. All of the running processes are held in a circular queue (the end of the queue loops back to its start), which is executed sequentially by the processor. Each process receives a slice of the processor's execution cycle after which control will be switched to the next process in the queue.

accessible from the host application solely through a fixed library interface, the ZBL-API. It has a fault tolerant design, i.e. run-time errors caused by programs running on the virtual machine, such as stack overflows or stack underflows. are recognised and result in the graceful degradation of the virtual machine. In effect this means that as soon as an error occurs, the game-bot program is terminated without affecting the execution of the virtual machine within its host application itself.

7.2.3.1 ZBL/0 Virtual Machine Instruction Set

The instruction set of the ZBL/0 virtual machine is that of a typical stack machine and not dissimilar from of P-code [Pemberton and Daniels 1982] (itself not too far removed from commercially available stack-based microprocessors). The original public release version of the ZBL/0 virtual machine (version 1.1, including a toolkit consisting of compiler, assembler and disassembler) [Zerbst et al. 2003] has a total of 33 instructions, although only 25 of these can actually be found in the bytecode generated by the ZBL/0 compiler, whereas the others allow for the creation of hand-optimised game-bot programs with higher code density. Of these 33 instructions, 5 are data instructions, 15 are arithmetic instructions, 4 are logic instructions and 9 are program flow control instructions (see Appendix C).

The intrinsic functions of ZBL/0, while implemented in the virtual machine, are not treated as virtual machine instruction but invoked as operands of a 'call' flow control instruction for executing a user-defined function.

An extended version of the virtual machine (version 1.2) used in our research [Anderson 2004] has added two additional instructions (one flow control instruction and one data instruction), allowing the extension of the system through plug-ins (see Section 7.2.5).

7.2.4 Extending a Game Engine with ZBL/0

If a virtual game world is to be populated with NPCs controlled by ZBL/0 programs, this requires the ZBL/0 virtual machine to be embedded in the game

engine, as well as the definition of a game-bot object that implements the intrinsic functions of ZBL/0 (and the provision of a suitable ZBL/0 game-bot program for the NPC). NPCs developed using ZBL/0 are purely reactive, i.e. their actions are direct reactions to an external stimulus such as a change in their perceived environment. In order to define the behaviour of an NPC, its domain knowledge, consisting of available sensor data and any information provided by the programmer within the behaviour definition program, needs to be combined with the controls that are available to the NPC, the latter being the ZBL/0 functions that execute game-bot actions. The behaviour of the game-bots that is perceived as intelligent therefore emerges from the combination of ZBL/0 command sequences in the ZBL/0 programs and the implementation of the functions of ZBL/0 within the host application.

The ZBL/0 virtual machine itself uses a predefined set of functions as an interface for communication with its host application. This allows it to execute the ZBL/0 functions that have been implemented within the host. The interface between the host application and the ZBL/0 virtual machine is the ZBL-API (see Section 7.2.4.1). The API takes the shape of a C++ interface for simple integration of the ZBL/0 system into other applications.

This interface to the ZBL/0 virtual machine provides game engines with the ability to associate NPC functionality with in-game functions for actions which would be expected to be performed by a player of these games, therefore allowing NPCs to compete with human players on a level playing field. Once a ZBL/0 program has been loaded into the virtual machine only a single function-call to the API is required in every execution cycle (usually a graphical frame) of the host application to execute the game-bot programs. The simplicity of the system lies in the fact that none of the game-bot functions are provided by the ZBL/0 system as such. Instead they need to be implemented within the game engine – the host application – and mapped to the corresponding intrinsic function identifier in ZBL/0. The game engine itself does all the work while the script only ties together the different game engine components that provide the NPCs with functionality.

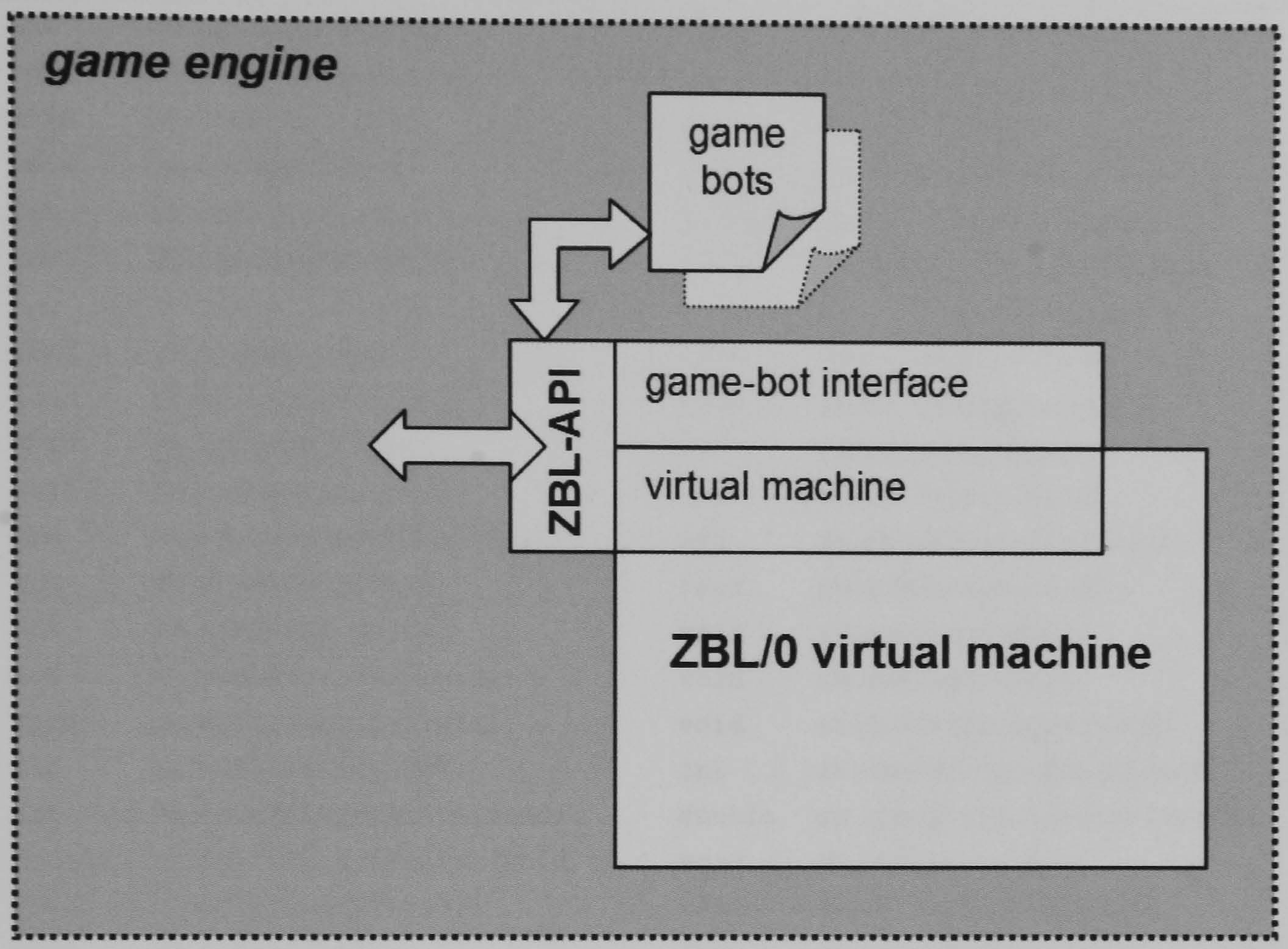


Figure 7.1: The interface between ZBL/0 virtual machine and host application.

7.2.4.1 The ZBL-API

The ZBL-API consists of two components (see Figure 7.1):

1. the game-bot interface for defining the functionality of the game-bots within the virtual game world

2. the virtual machine interface for the execution of ZBL/0 bot programs within the game engine

The core of the ZBL-API is part of the game-bot interface for the definition of game-bots, which is defined as an abstract base class (zblbot) which greatly simplifies the implementation of game-bots within C++ based game engines. The function bindings between the host application and game-bots running on the ZBL/0 virtual machine are realised using the multiple-inheritance functionality of

```
int     zb_checkBlocked(int);          void    zb_doCrawl(void);
int     zb_checkDanger(void);          void    zb_doDie(void);
void    zb_doDuck(void);               int     zb_doFace(int);
void    zb_doFind(int);                void    zb_doFire(void);
int     zb_mdfFront(void);             double  zb_checkHealth(void);
int     zb_checkIdle(void);            void    zb_doInitialize(double,
                                                       double,double);
void    zb_doJump(void);               void    zb_doJumpBack(void);
void    zb_doJumpLeft(void);           void    zb_doJumpRight(void);
void    zb_doJumpUp(void);             int     zb_mdfLeft(void);
void    zb_doMemorize(int);            int     zb_mdfObject(void);
int     zb_checkObjectAhead(void);     int     zb_checkObstacle(void);
int     zb_checkOwns(int);             void    zb_doRespawn(void);
int     zb_mdfRight(void);             void    zb_doSpawn(void);
int     zb_checkSpawned(void);         void    zb_doStep(void);
void    zb_doStrafeLeft(void);         void    zb_doStrafeRight(void);
int     zb_mdfTarget(void);            int     zb_checkTargetAhead(void);
int     zb_checkTargetAlive(void);     double  zb_checkTargetArmour(void);
double  zb_checkTargetHealth(void);    void    zb_doTurn(double);
void    zb_doTurnLeft(void);           void    zb_doTurnRight(void);
void    zb_doUse(int);                 int     zb_checkUsing(void);
```

Table 7.3: Game-bot interface methods of the ZBL-API (class zblbot).

the C++ programming language [Stroustrup 1997]. A game-bot class from which NPCs can be instantiated is created by inheriting player functionality from the game-bot interface of the ZBL-API and a player-class in the application which should provide game-bots with the same level of control that a human player would have. An implementation of the abstract functions from the game-bot interface then allows ZBL/0 programs in the virtual machine to control a game-bot character in the application. The game-bot interface of the ZBL-API consists of 44 methods (for prototypes see Table 7.3). These methods map directly to the standard functions of the ZBL/0 scripting language. The exact implementation of these methods depends on the host application into which the game-bots are supposed to be integrated. Consequently the implementation will vary from game engine to game engine.

Apart from these methods, the abstract game-bot interface class contains a number of additional attributes and methods which themselves are not part of the bot interface but which are nevertheless important for the correct execution of game-bot programs. They are used in the implementation of the methods of the game-bot interface that map to ZBL/0 functions whenever that function causes a game-bot to begin an action which may take some time (more than a single virtual machine execution cycle) to finish and if the execution of subsequent instructions to the game-bot should be suspended while the action has not finished, such as an animated movement that may be executed over several frames. This blocking of processes in the virtual machine is similar to the yield operation for coroutines found in the language Lua [Ierusalemschy et al. 2007]. This is achieved through the zblbot class attribute 'zb_busy' which holds the blocked state of a game-bot and determines if a bot process is active or suspended within the virtual machine while it is waiting for an earlier action to finish. To prevent errors caused by unintended manipulation of the 'zb_busy' flag, direct access to this attribute of the zblbot class should be avoided, which is why the class provides three methods for this task. The current blocked state of a game-bot can be queried using the 'zb_getBusy' method. The 'zb_setBusy' method suspends the execution of a bot program by setting the blocked state of an active game-bot to be true. To continue program execution and to reactivate a blocked game-bot the 'zb_unSetBusy' method must be called, i.e. it is essential to unblock the bot process to end the suspension of the execution of the bot program once the game-bot no longer needs to be blocked. Usually this would be done when an action that required the blocked state of the game-bot to be set to true has finished. If the programmer implementing the game-bot fails to do so, this omission will not only retain the suspension of the bot process, effectively breaking the program, but it may also lead to a time wasting overhead within the virtual machine due to unnecessary switching between game-bot processes. If that should happen and a suspended game-bot has not been reactivated for some considerable time, suggesting that there is a problem with the implementation of the NPC functionality, the ZBL/0 virtual machine will handle this situation as a run-time error and the offending bot process will be stopped.

```
double      zbl_getVersion(void);
char       *zbl_getVersionString(void);
int         zbl_addProcess(char *filename,zblbot *bot);
void        zbl_removeProcess(int pID);
int         zbl_replaceProcess(int pID,char *filename);
void        zbl_resetProcess(int pID);
int         zbl_replaceBot(int pID,zblbot *bot);
void        zbl_setPriority(int pID,int pr);
int         zbl_run(void);
int         zbl_getErrors(void);
zbl_error_t zbl_nextError(void);
zbl_error_t zbl_peekError(void);
```

Table 7.4: Virtual machine interface methods of the ZBL-API (class zbl_vm).

The second component of the ZBL-API is the virtual machine interface that provides 12 methods (for prototypes see Table 7.4) that allow the loading and execution of ZBL/0 programs in the virtual machine. Embedding the ZBL/0 virtual machine into a game engine using the methods of the virtual machine interface is relatively simple. Apart from the instantiation of the virtual machine by creating an object of the virtual machine class (zbl_vm), all a minimum implementation for a game-bot requires is the creation of a "process" for the virtual entity on the virtual machine (using the method 'zbl_addProcess'), as well as an invocation of the scheduler for every update cycle (by calling the scheduling method 'zbl_run').

7.2.4.2 Using the ZBL-API

Given a game-bot class as defined below (gameBot), inheriting from the ZBL-API's bot interface as well as a player class (gamePlayer), providing the game-bot with the functionality available to a human player, any object that is created as an instance of this derived class is an NPC that can be used by the ZBL/0 virtual machine.

```
#include <zblbot.h>
...
class gameBot : public gamePlayer, public zblbot
{
    ...
};
```

For an instance of this game-bot to be controlled by a ZBL/0 program, the ZBL/0 virtual machine needs to be instantiated:

```
#include <zblvm.h>
...
{
    zbl_vm  myVirtualMachine; // instance of the ZBL/0 VM
    gameBot myGameBot;        // instance of a game-bot
    ...
```

The ZBL/0 game-bot program that is to be executed by the virtual machine has to be loaded into the virtual machine and must be mapped to the game-bot object for the NPC (using the 'zbl_addProcess' method of the virtual machine).

```
    ...
    myVirtualMachine.zbl_addProcess("botprogram.zbp",
                                    &myGameBot);
    ...
```

Once this set-up of the virtual machine has been completed, the 'zbl_run' method of the virtual machine should be called once during every update-cycle (once per rendered frame) of the host game engine. This method processes the process list within the virtual machine and executes the ZBL/0 game-bot programs.

```
    ...
```

```
    myVirtualMachine.zbl_run();
    ...
}
```

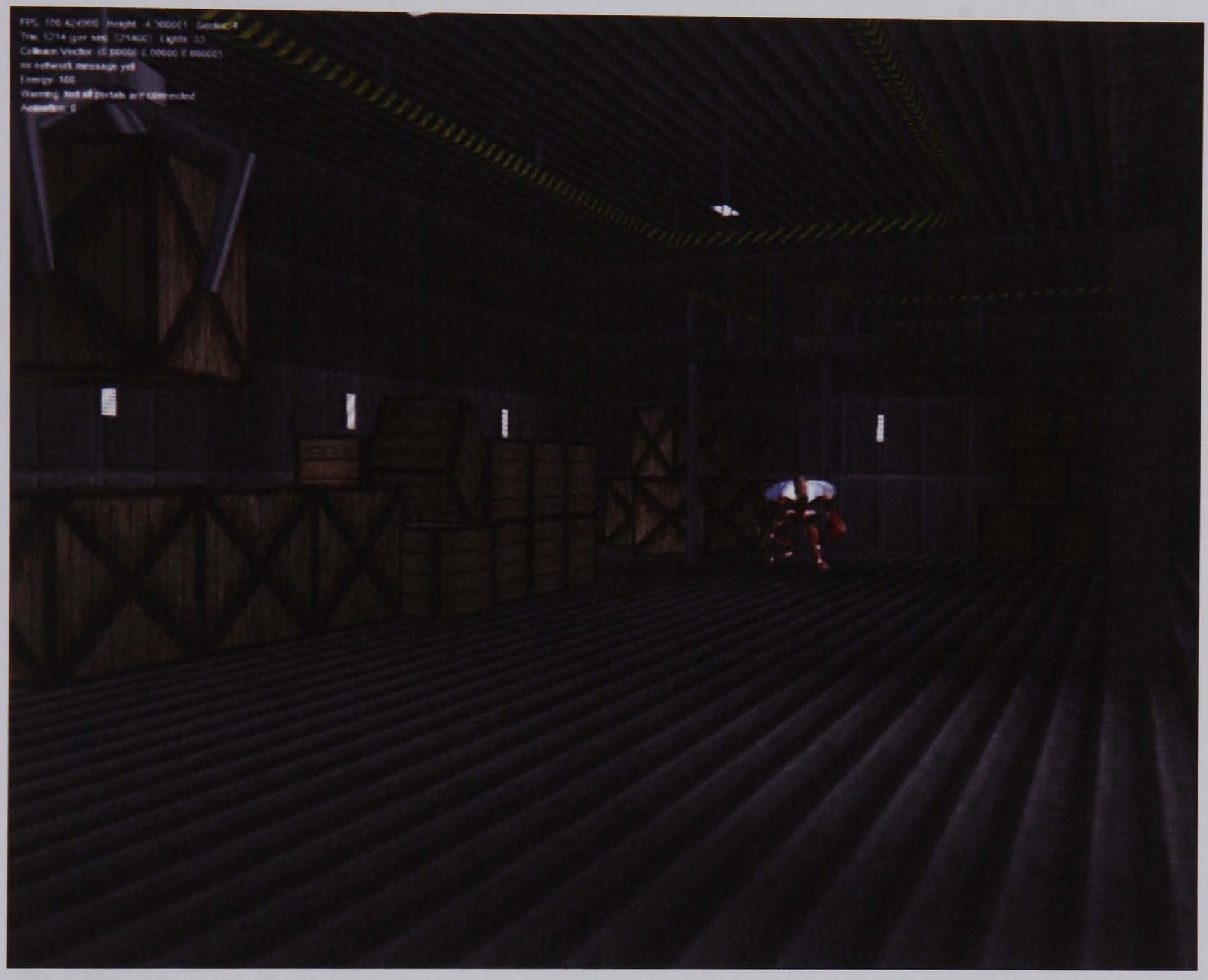


Figure 7.2: A ZBL/0 game-bot patrolling a warehouse.

This is all that would be required for embedding the ZBL/0 virtual machine into a game engine. Any additional operations, such as querying of virtual machine run-time errors or more complex process management are optional, i.e. not required for adding ZBL/0 to a game engine.

7.2.4.3 Designing a Simple ZBL/0 Game-Bot

To demonstrate the usage of ZBL/0 we can look at the development through step-wise refinement of a simple NPC that patrols a warehouse (see Figure 7.2). Written in ZBL/0 a simple version of the program would look as follows:

```
{
  while alive do
  {
    if blocked front then
    {
      # 180 degree turn
      turn_left;
      turn_left;
    };
    else
      step;
  };
}.
```

The above is a short, simple program, storing the whole of the NPC behaviour definition within a single routine. If it were any more complex, a bot-program written like this would quickly grow to a size that would make it unmaintainable. To prevent this it makes sense to provide the script with some structure and to break the program up into separate functions as shown in the next iteration of the program:

```
function turn180;
{
  # 180 degree turn
  turn_left;
  turn_left;
};
```

```
function patrol;
{
  if blocked front = 0 then # if path in front is blocked
    turn180;                # turn around
  else                      # turn around
    step;                   # otherwise
};

{
  while alive do
    patrol;
}.
```

This version of the program shows the use of functions in ZBL/0, the names of which (turn180, patrol) were chosen to be self-explanatory. An NPC controlled by the above script will patrol the area of the virtual game world in which it has been placed. If it encounters an obstacle it will turn around using a left turn and return to its starting point on the same path. While this behaviour might be acceptable for an incidental that paces an area up and down, it would make little sense for a game-bot that is supposed to hold watch and guard an area. In the latter situation the NPC should ensure that its back is always covered and turned away from any danger, i.e. preferably it should keep its back to a wall when turning to prevent being ambushed from behind.

An improvement to the turn180 function, solving this problem, should ensure that a scripted NPC only turns in a given direction if it does not expose its back in the process:

```
function turn180;
{
  if !blocked left then # if no wall to the left
  {                     # turn around left
    turn_left;
```

```
    turn_left;
  };
  else                    # otherwise
  {                       # turn around right
    turn_right;
  };
};
```

The resulting NPC will patrol an area in the game world as long as it is "alive", however it will mostly ignore its environment, except for obstacles in its path. It would be unable to sense an opponent and therefore unable to defend itself against attack. To rectify this, the behaviour definition program needs to be extended by a conditional selection in the main program that only allows the NPC to continue its patrol if there is no danger nearby. For the final iteration of this program the game-bot should switch from patrol into skirmish mode in the case of looming danger:

```
# main program for the combat bot
{
  while alive do
  {
    if !danger front then # if no enemy in sight
      patrol;             # patrol the perimeter
    else                  # if enemy nearby
      skirmish;           # attack!
  };
}.
```

The above listing shows the modified main function of the behaviour definition script. If there is no danger, the game-bot follows its patrolling behaviour by calling the function 'patrol', however if there is danger, the NPC enters combat behaviour by calling the 'skirmish' function which executes a simple attack

pattern:

```
function skirmish;
# function skirmish - aim at target and attack
{
if target_ahead then # if clear sight of target
{
  fire;                  # fire salvo 1
  fire;                  # fire salvo 2
  fire;                  # fire salvo 3
  step;                  # advance
};
else                     # otherwise
{
  if !blocked front then # if the path is clear
  {
    face_target; # turn towards / aim at target
    if target_alive then # still breathing?
    {
      fire;              # fire once
      step;              # advance
      fire;              # fire another shot
    };
    else                   # otherwise
      step;              # step forward
  };
  else                     # if no clear shot
    backstep;            # retreat one step
};
};
```

Once this script has been compiled using the ZBL/0 compiler, the resulting game-bot is a fully functional NPC, able to defend itself against approaching

enemies, that could be used as a "guard" character in a computer game.

7.2.5 ZBL/0 Extensions

We have used the ZBL/0 system as a test-bed for interfacing behaviour-definition systems with computer games. The primary development was that of a plug-in architecture that was implemented to deliver a partial solution to problems caused by the lack of extensibility of the core ZBL/0 system. This plug-in system provides the mechanisms for adding additional functions and constants to the virtual machine, which could be extended to add additional operators as well. Plug-ins are loaded into the ZBL/0 compiler, as well as the virtual machine. To use plug-ins, the ZBL/0 language has been expanded by an additional statement for loading a specific plug-in for use. If this statement is used in a ZBL/0 program, the compiler loads the plug-in and queries it for a list of the identifiers of functions and constants contained within, so they are known to the compiler and can be used in the ZBL/0 program. Any calls to the plug-in's functions are then stored in the program's bytecode as some sort of position independent code, relative to the plug-in used. For this the bytecode representation has been extended to optionally store information on used plug-ins. This information is then used when the program is loaded by the ZBL/0 virtual machine which then loads these plug-ins into the virtual machine. When called from the game-bot program, the virtual machine passes control of the stack of the game-bot process to the plug-in with a request to execute the appropriate function in the plug-in. We believe that this method for extending the ZBL/0 virtual machine is a feature that provides a useful mechanism for adding extensibility to any behaviour definition system.

Apart from the additional virtual machine instructions for using the plug-ins, the main change to ZBL/0 system was the extension of the functionality of the virtual machine and consequently additions to the virtual machine interface, adding not only methods for dealing with the plug-in extensions, but also utility functions to help with debugging ZBL/0 programs, such as the facility to create a pretty printed stack dump for running bot processes. Other changes have involved a modification of the scheduler to achieve dynamic load balancing of the virtual machine.

7.2.6 Concluding Remarks on ZBL/0

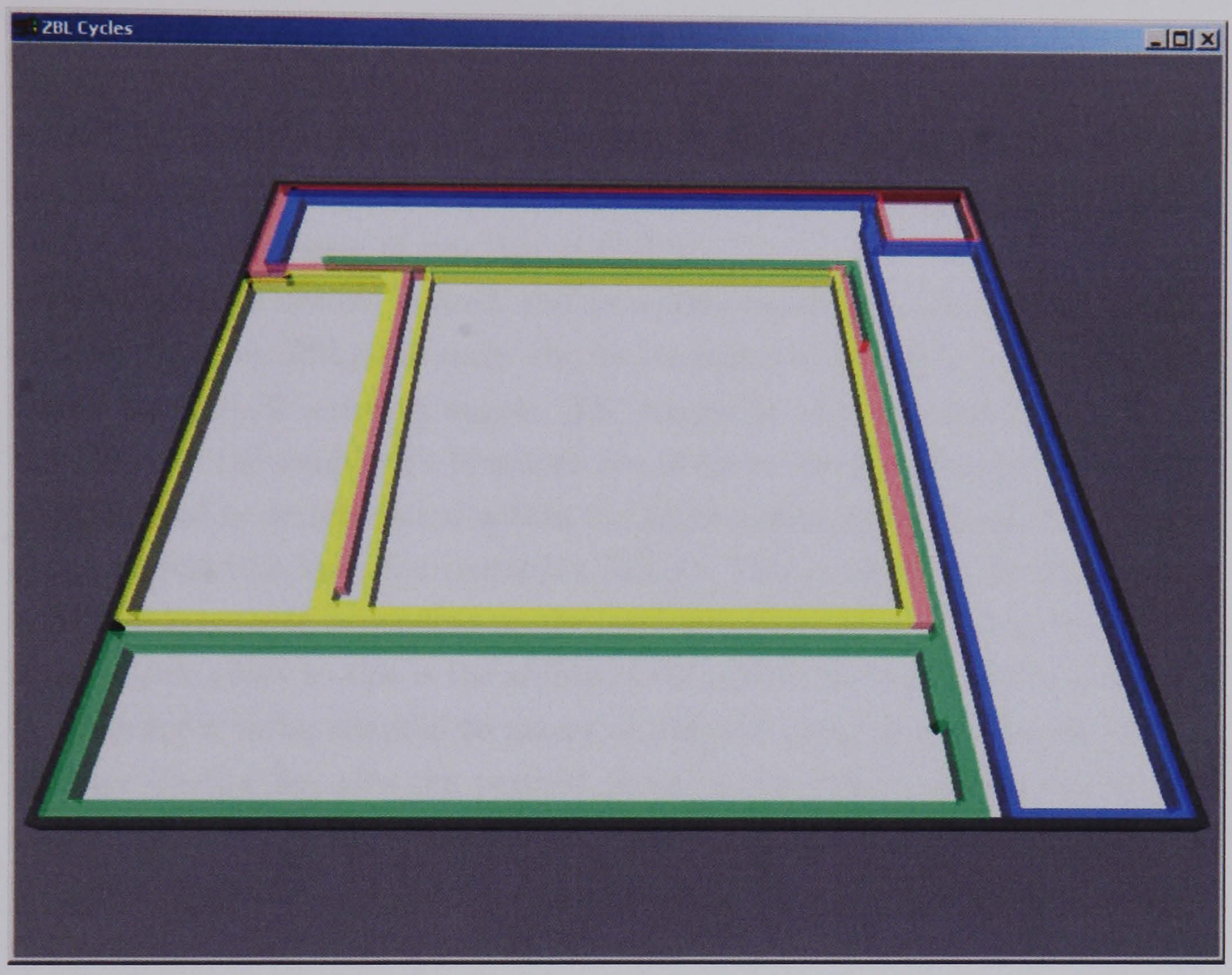


Figure 7.3: ZBL/0 game-bots in a "light-cycle race"

ZBL/0 [Anderson 2003b] is a very simple scripting language for the definition of NPCs (tactical opponents, incidentals, team-mates and even observers – see Chapter 2, Section 2.2) in the virtual worlds of FPS or possibly third person perspective action games. The ZBL/0 system consists of a compiler for game-bot programs (the very NPCs) written in the ZBL/0 language and a robust virtual machine that can be embedded into any C++ based game engine.

While its purpose is that of a behaviour definition language, the ZBL/0 scripting language does not really conform to the requirements for behaviour definition languages that we have identified (see Chapter 5, Section 5.2.1). ZBL/0 does not have any AI specific data types or operators and it is a strictly procedural

programming language without any object orientation. The syntax and structure of ZBL/0, however, allow for the creation of state machines in a similar manner to those defined in any generic programming language, such as C. Nevertheless, the ZBL/0 virtual machine matches several of the requirements for a behaviour definition run-time system, i.e. it provides a platform independent embeddable module (so far implemented under Windows and Linux), a small execution overhead and a high degree of run-time stability.

If the ZBL/0 system is used, the seemingly intelligent behaviour of NPCs is not generated by ZBL/0 as such: the functionality of ZBL/0 is not implemented within the ZBL/0 scripting engine. The simplicity of the system lies in the fact that none of the language's functions are provided by the language as such but must instead be implemented within the game engine and mapped to the corresponding function identifier (name) in ZBL/0. This means that the functionality of ZBL/0 is entirely dependent on the implementation of the host application. A positive side effect to this is the ability of the system to transcend its limitations by allowing it to be adapted to games of different game genres (see Figure 7.3), however therein lies also the weakest point of the ZBL/0 system as any NPC script, no matter how well designed, cannot perform well if the NPC related functions of the game engine do not work well. On the other hand, this system allows the designer to create effective NPCs through the combination of a range of relatively simple functions.

As such, we have used the ZBL/0 system to explore various system architectures for integrating virtual machines into applications – simple games and more complex game engines – that allow scripts to be executed and interpreted in real-time.

We have identified several other parts of the ZBL/0 system that have shown weaknesses in the original design concept which we intend to address with our current and future work. For instance the lack of extensibility provided by the method in which function bindings are implemented in ZBL/0 has convinced us that a different approach will have to be used for more generic behaviour definition systems. While the method used makes it very easy for the virtual machine to execute functions within the host application it also limits the extensibility of the ZBL/0 system and the use of intrinsic functions results in the main cause of

inflexibility of the ZBL/0 system, as the type and number of the functions that can be registered with the virtual machine is fixed by the ZBL-API. Related to that we believe that an implementation using external libraries to provide the core language with functionality would have made the system much more extensible and flexible.

7.3 Educational Programming Language C-Sheep

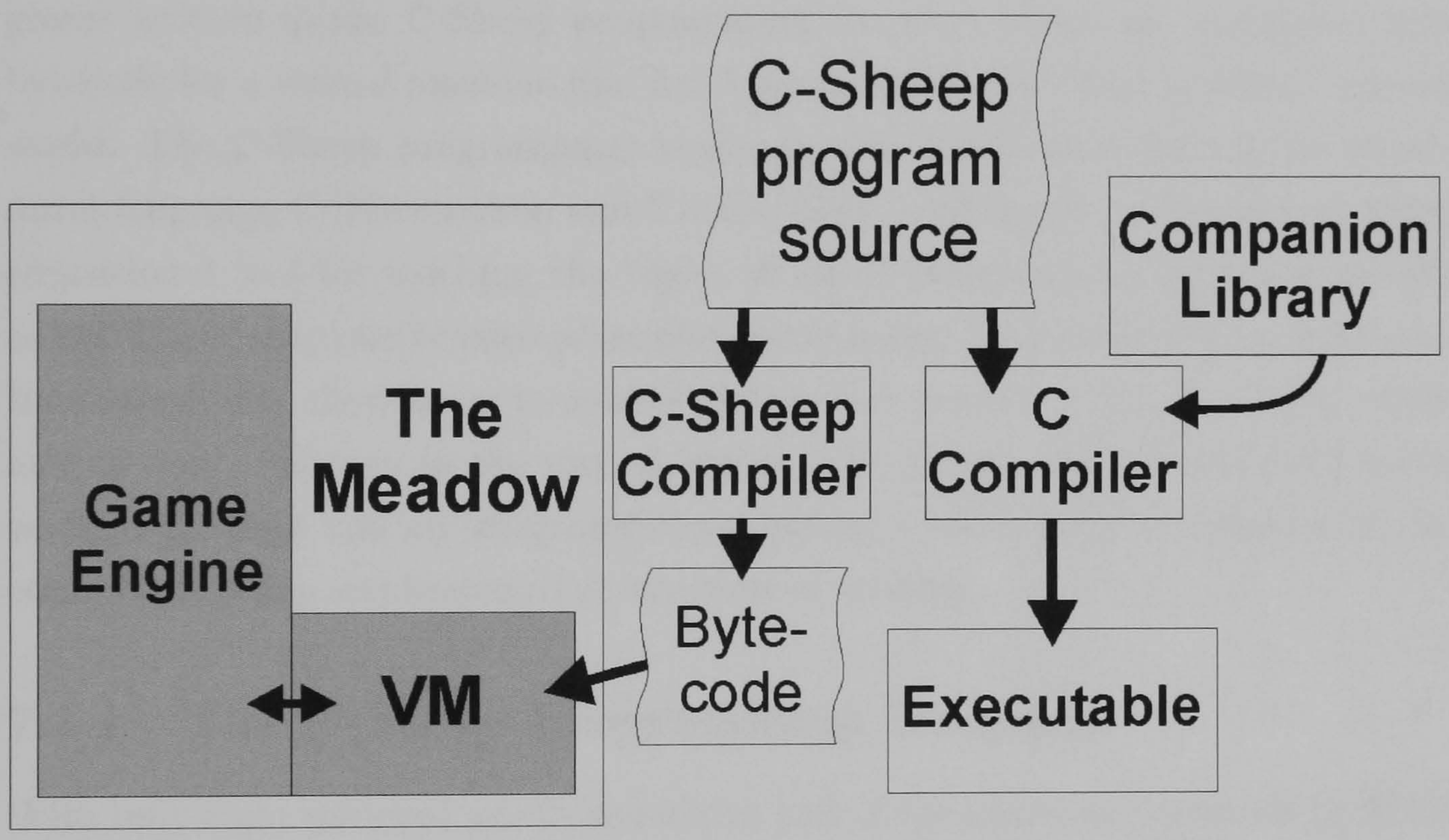


Figure 7.4: Components of the C-Sheep system.

Inspired in part by existing educational mini-languages (toy languages) used in teaching [Brusilovsky et al. 1997], as well as our own ZBL/0 behaviour definition language (see Section 7.2 above), we have developed C-Sheep, a Regular Script type (*ST3b* – see Chapter 6, Section 6.1.2) behaviour definition language, as a teaching tool for computer science education and the introductory computer programming sequence, using a state of the art rendering engine, usually found

in entertainment systems, to motivate students to spend more time programming (see Figure 7.4) [Anderson and McLoughlin 2007].

Most traditional mini-languages are now severely outdated [Anderson and McLoughlin 2006] and consequently fail to maintain the interest of students. The C-Sheep system aims to address this issue by enhancing the teaching tool with the visual gimmickry of modern computer games, which allows programs to provide instant visualisation of algorithms in a visually appealing, game-like virtual environment. This environment, called "The Meadow", provides a game world inhabited by an NPC-like virtual entity, namely a sheep that can be controlled by C-Sheep programs. The system as such consists of a compiler for entity programs written in the C-Sheep programming language which are translated into bytecode for a virtual machine that has been embedded in "The Meadow" virtual world. The C-Sheep programming language was closely modelled on the procedural language C [Kerninghan and Ritchie 1988], making the C-Sheep system an educational tool for teaching the basics of the C programming language as well as the basic computer science principles encountered in structured programming. Its instructions allow users to control the actions performed by the sheep entity and to query changes in the virtual world. The system itself is still very much work in progress and an integrated development environment with an on-the fly compiler is being implemented at the time of writing.

7.3.1 The C-Sheep Programming Language

With only eight reserved words and about half of the operators available in ANSI C, the C-Sheep programming language is a small, yet fully compatible subset of the ANSI C programming language (see Figure 7.5), i.e. programs written in C-Sheep should be compilable on any ANSI C compiler. Through the provision of a "companion library" that mirrors the C-Sheep instructions for the sheep entity in the virtual machine, it is even possible to compile frozen executables of C-Sheep programs using an off-the-shelf ANSI C compiler.

The design of the C-Sheep subset of C was guided by some of the introductory language design principles proposed by [McIver and Conway 1996], with

the provision of a set of non-overlapping language features to prevent "information overload" among programming novices considered as especially important. C-Sheep implements the C control structures that are required for teaching the basic computer science principles encountered in structured programming, these being the (unconditional) sequence, conditional statements and loops [Böhm and Jacopini 1966]. In terms of syntax, these control structures are the block, if and if-else alternatives, as well as while and do-while loops. Additional control structures such as the counting for-loop, the multiple-selection switch-statement or the ternary selection operator '?:' were omitted from the language's syntax to minimise any confusion that these overlapping control structures could cause for a novice programmer.

In addition to the predefined sheep control instructions in the standard libraries (see Section 7.3.1.1 below), C-Sheep also allows the definition of subroutines – like functions in C, first level order functions – which can be called recursively. As is the case with C functions, functions in C-Sheep can return values of the available variable data types (unless declared as 'void', i.e. typeless) and receive parameters.

C-Sheep variables are strongly typed and can either be integers of the type 'int' or real numbers of the type 'float'. These can be declared globally within the body of a C-Sheep program or locally at the start of functions and blocks of C-Sheep code. Constant values in C-Sheep programs can be defined using the '#define' directive as used in C programming language preprocessors, but unlike the text-substituted symbolic constants this would create in C, constants in C-Sheep are real constant values, with the benefit of type-safety. The language allows the definition of C-style constant strings (characters between opening and closing quotation marks) for printing, whose contents are not variable and cannot be changed during program run-time, however they can be used as format strings that can be used to create variable output.

7.3.1.1 The C-Sheep Standard Libraries

The C-Sheep standard libraries (see Table 7.5) provide a number of general purpose functions and sheep-specific functions (functions for controlling sheep entities

```
void   abort(void);              void  exit(int);
int    rand(void);

float  sqrt(float);

void   printf(char*,...);

void   pause(void);              void  initialise(int,int);
void   step(void);               void  backstep(void);
void   turn_left(void);          void  turn_right(void);
int    blocked(int);             int   found(int)
float  query(void);
```

Table 7.5: C-Sheep standard functions.

in the virtual environment). The latter are reminiscent of the intrinsic functions of the ZBL/0 scripting language, but whereas in the case of ZBL/0 the programming language was designed specifically as an educational tool with the definition of NPC (Non-Player Character) behaviour in First Person Shooter (FPS) games in mind [Zerbst et al. 2003], the C-Sheep language only required basic control of the virtual entity's movements. This is why only a few of ZBL/0's functions and procedures remain present in some form in the C-Sheep language. The instructions of C-Sheep reflect those available in other educational languages, such as "Karel the Robot" [Pattis 1981], i.e. "sheep functions" for controlling the sheep (by exposing sensor information and instructions for the sheep entity) in the virtual world, declared in the language's 'sheep.h' header file. Some of these sheep-specific instructions allow the querying of states in the virtual world, such as the current state of the weather. These world states can be altered interactively by the user (while C-Sheep programs are running in "The Meadow"), adding a separate layer of interactivity to the learning game. By instigating a state change in the virtual environment, the user can cause different sections of C-Sheep programs to be executed, allowing experimentation with different behaviours of the sheep entity from within the same C-Sheep program.

The general purpose function set implemented in the C-Sheep system provides several general purpose functions and associated symbolic constants found in the

ANSI C standard libraries, declared in the 'stdlib.h' standard header file. In a similar manner, C-Sheep's maths function 'sqrt' for calculating the square root of a given value is taken from the 'math.h' standard header file of the ANSI C standard libraries and the string output function 'printf' which behaves similar to the one defined in the 'stdio.h' standard header file of the ANSI C standard libraries, i.e. taking a (format) string as its first parameter, followed by a list of optional values for printing.

To access the C-Sheep library functions, C-Sheep programs must include the headers containing the function prototypes. This is done only to introduce novice programmers to the concept of code modularisation and libraries, i.e. all of the functions are disguised as library functions and C-Sheep programs must contain an include statement to (supposedly) parse the function prototypes in order to be able to call the functions, while internally these functions are actually intrinsic to the virtual machine.

7.3.2 The C-Sheep Virtual Machine

The virtual machine used in the engine is an improvement on the latest version of the ZBL/0 virtual machine [Anderson 2004] (see Section 7.2 above) without the experimental plug-in architecture, i.e. the virtual machine uses a parallel stack-based architecture and allows more than one program to run simultaneously through multi-tasking, although in the current implementation of the system usually only one virtual entity inhabits the game world. The virtual machine retains full backwards compatibility to bytecode compiled from programs created using the ZBL/0 core language, i.e. all of the instructions of the ZBL/0 virtual machine exist in the virtual machine for C-Sheep programs. This can easily be demonstrated with ZBL/0 programs that can be executed within the virtual machine, in which case the NPC controlled by the ZBL/0 program in "The Meadow" is the sheep entity (see Table 7.6). Character strings for output (see 'printf' function in Section 7.3.1.1 above) are located at the end of bytecode executable files. To allow for their output, the virtual machine stores them in constant sized memory segments that are separate from the code segments and the stacks of processes, from where the programs can access them.

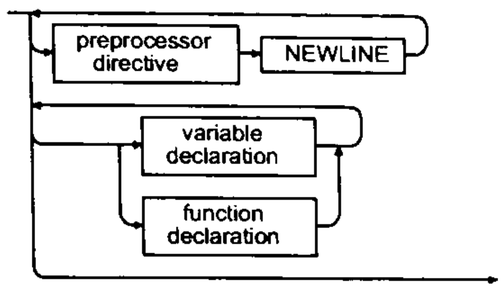
ZBL/0	C-Sheep

var y;
{
  y=1;
  while y=1 do
  {
  if blocked front = 0 then
  {
  step;
  };
  else
  {
  turn_right;
  };
  };
}.
	#include<sheep.h>
	
	main()
	{
	  while(1)
	  {
	  if(!blocked(FRONT))
	  step();
	
	  else
	  turn_right();
	
	  }
	  return 0;
	}

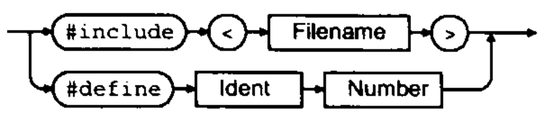
Table 7.6: A simple C-Sheep program in comparison to an equivalent program written in ZBL/0: if the path of the sheep entity is blocked, it will turn right, otherwise it will take a step forward.

The virtual machine of the C-Sheep system provides additional intrinsic functions mirroring the C-Sheep standard libraries' general purpose functions, maths functions and those sheep-specific functions that have no equivalent in ZBL/0. The intrinsic functions of the ZBL/0 language that do not have an equivalent to the functions found in the C-Sheep standard library are present in the virtual machine, but do not have any functionality associated with them, i.e. calling them will have no effect on the sheep entity inhabiting "The Meadow".

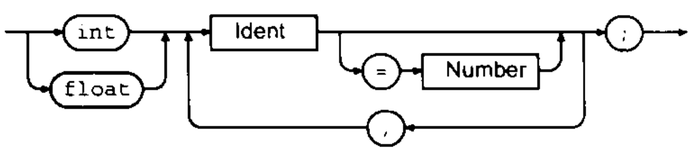
program:



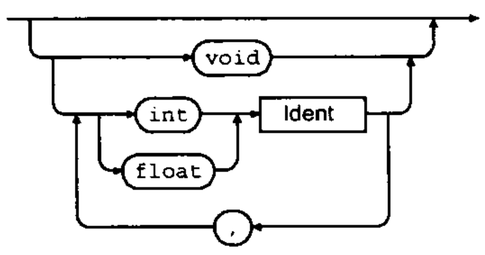
preprocessor-directive:



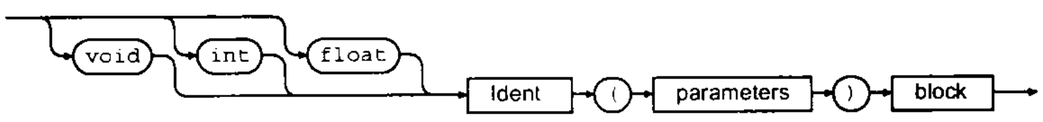
variable-declaration:



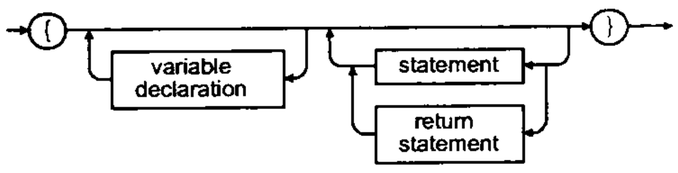
parameters:



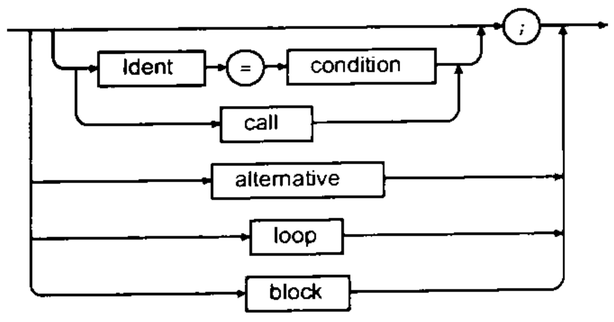
function-declaration:



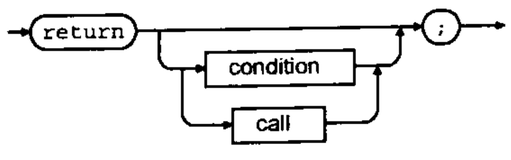
block:



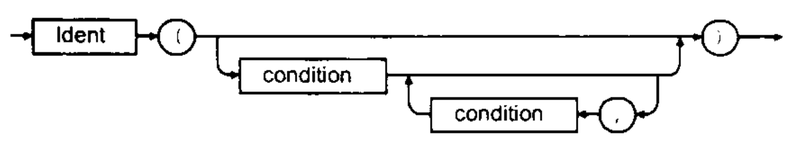
statement:



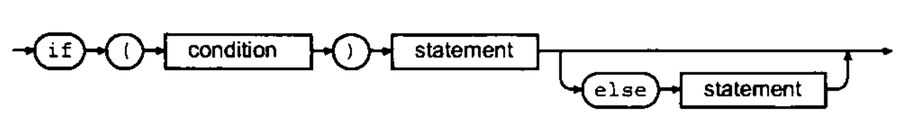
return-statement:



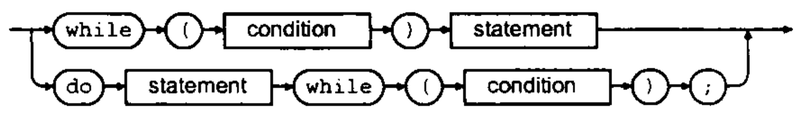
call:



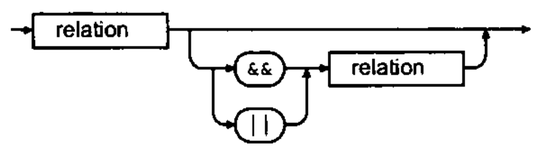
alternative:



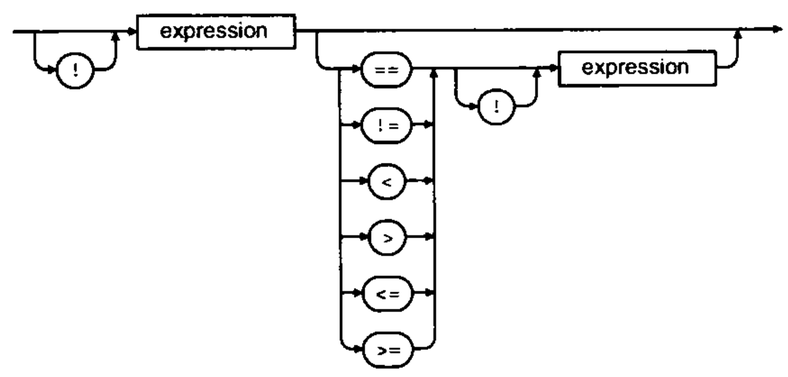
loop:



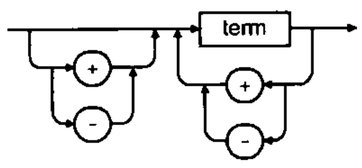
condition:



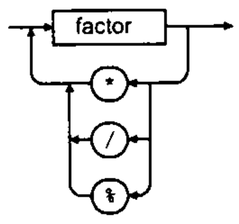
relation:



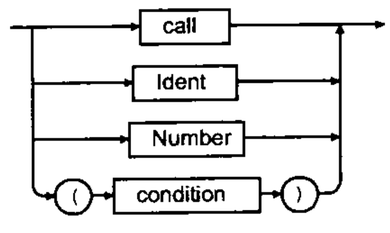
expression:



term:



factor:



NEWLINE	terminal symbol: the operating system's newline symbol (typically a combination of line-feed and carriage-return control characters)
Ident	standard identifer (first character must be a letter, followed by a sequence of characters that may be letters, digits or the '_' symbol, ending with a whitespace)
Number	numerical value that can be any integer or the decimal representation of a real number
Filename	constant string encoding a C-style header file filename (*.h)

Figure 7.5: C-Sheep Syntax.

7.3.2.1 C-Sheep Virtual Machine Instruction Set

While the C-Sheep virtual machine's ZBL/0 subset is identical to the 33 instructions of the ZBL/0 virtual machine's instruction set, some of the subset's instructions can take additional parameters that are C-Sheep specific and that are currently only generated by the C-Sheep compiler. This becomes obvious in instruction sequences that deal with function parameters for user-defined functions which needed to be integrated into the virtual machine while still allowing ZBL/0 programs to run, although function parameters did not exist in ZBL/0. The virtual machine as such has a total of 41 instructions, providing major improvements over ZBL/0 as the instruction set of the virtual machine includes instructions related to the use of constant character strings, as well as facilities for the use of pointers and for the creation of aggregate data types (arrays and record structures). While much of this is unused as it exceeds the requirements of the C-Sheep C language subset, the provision of this rich feature set provides upwards compatibility to possible future revisions of the system, as well as source language independence.

The C-Sheep language contains several data types for numerical values, however, this is not reflected in the virtual machine's instructions, but rather emulated by the compiler, whereas internally actually only one type is used, as is the case with ZBL/0.

7.3.3 Concluding Remarks on C-Sheep

While not strictly speaking a behaviour definition language according to our definition (see Chapter 5, Section 5.1), because despite its presence in a game-like environment the virtual entity controlled by C-Sheep programs is not a classical NPC (see Chapter 2, Section 2.2), C-Sheep nevertheless has a lot in common with other behaviour definition languages. This is mainly due to the C-Sheep run-time system, which is shared to a large extent with our ZBL/0 behaviour definition system (see Section 7.2 above).

C-Sheep is a mini-language-like programming language for computer science education with a run-time system embedded within a state-of the art 3D computer game-like virtual environment.

A feature in which C-Sheep differs from other educational systems that aim to provide an environment with minimal complexity, is that C-Sheep allows the declaration and use of variables, which some educators might consider undesirable, as this increases the complexity of the language. There are, however, problems with variable free languages that can occur at the moment of transition to a real-world system, as identified by Untch [1990]. If an educational programming language has variables, the migration to a real-world language can be delayed and the transition to the real-world system will come as less of a shock to students, or as Untch says, "the students always add to their stock of knowledge and never need to relearn (or unlearn) something" [Untch 1990]. Other possible benefits to the inclusion of variables in an educational language are that they allow the learner to track object's histories (as non-visual states), i.e. as counters [Dann et al. 2000].

The C-Sheep system, including the C-Sheep programming language and "The Meadow" virtual environment are ongoing projects, and continue to be developed further. In its current implementation, the language's compiler is kept separate from the run-time system (although an integrated on-the-fly compiler is in development). This is where the system differs from the more integrated development environments of other educational systems. While this use of an external compiler slightly complicates the use of the system it also creates greater flexibility by freeing "The Meadow" from being bound to the C-Sheep language (source language independence), making it targetable by compilers for different languages, i.e. a Java based J-Sheep or Pascal based P-Sheep could be created with relatively little effort.

Part III

A Behaviour Definition Language

Chapter 8

NPC Behaviour Definition Language AvDL

The main body of our work consists of the development of a behaviour definition (programming) language for the creation of believable NPC intelligence [Anderson 2005a]. This Avatar Description Language (AvDL) provides the core of a generic, modular and easily extendable system for the definition of believable intelligent game character behaviour. Since its original conception, the language specification for AvDL has undergone a series of step-wise refinements, some of which are detailed in the discussion of language features and syntactical elements below (see Section 8.2), whereas some are discussed later in this thesis (see Chapters 9 and 11).

This embeddable Regular Script type (*ST3b*) scripting language, a substantial subset of which we have already implemented (see Chapter 9), provides for the definition of deterministic, as well as goal-directed behaviours, allowing the system to be used for different purposes and with a wide range of computer game related applications.

Like most currently available scripting systems, the AvDL run-time environment employs a stack-based architecture (see Chapter 10) and while the AvDL's primary purpose is the definition of NPC behaviour, the system's large degree of flexibility and extensibility does not necessarily limit it to this task.

8.1 Towards a Better System for Defining Computer Game AI – Rationale for the AvDL Scripting Language

Over the past decade we have seen that the development of sophisticated 3D modelling and animation programs have given rise to new methods and increased realism in film, animation and computer games. This and more powerful hardware have led computer games to evolve from existing in two dimensions only into the third dimension, spawning a range of new computer game genres.

However, whereas computers get faster, unfortunately programmers themselves do not. The use of a generic system for the definition of NPC behaviour would speed up the development of computer controlled NPCs, simplifying the game development process and resulting in a reduction of the workload of game programmers. Additionally, the improvement of the game development process through close integration of this system with the tools and libraries that are already used for creating computer games, such as level editors or 3D content creation applications, would be highly desirable. While making it easier for a game production team to meet its budgetary constraints, such a system would also make game development more cost effective by allowing parallel development, i.e. the creation of NPCs independent from the main game source code.

We believe that the introduction of this type of generic behaviour definition system would be able to contribute to the continuing evolution of computer games and open up new avenues for greater realism within the virtual game worlds.

The development of the AvDL behaviour definition system has been guided through the understanding of current computer game AI techniques (see Chapters 2 and 4) and their application to modern computer games. The insights that we have gained during our review of the available literature and the design of our experimental ZBL/0 scripting system (see Chapter 7, Section 7.2) have greatly influenced our design approach towards our system for defining NPC behaviour. Compared to existing AI middleware systems our solution should allow for a much smoother integration of the behaviour definition system into game applications as it provides a combination of all of the different components for creating NPCs (see

also Chapter 4, Section 4.1) within a unified architecture that binds these modules together. Consequently the need for users to change between different APIs and applications to achieve their goals will be greatly reduced. This also means that there should be considerably fewer restrictions imposed by compatibility problems between different systems.

8.2 The AvDL Programming Language

AvDL is a universally applicable extension programming language for the definition of artificial behaviour for virtual entities, i.e. creatures and characters, in computer games and potentially in computer animation. AvDL is defined using an LL(1) grammar (see Appendix D). AvDL programs are executed in a virtual machine that can be integrated into any game engine.

AvDL is not bound to a single genre of computer game or NPC. To maximise the language's potential for NPC behaviour definition it allows for different programming styles, i.e. AvDL supports the object oriented programming paradigm but also allows the creation of procedural programs as well as event based programming (see also Chapter 6, Section 6.1.2).

For this, the language provides various means for defining NPC program flow and some NPC AI related data types and operators.

The system's flexible structure allows for high extensibility, therefore making AvDL usable for a wide variety of different tasks. This makes AvDL a lot more similar to an implementation language than to what is often considered a scripting language (see also Chapter 6) which will become more apparent when the exact features of AvDL are examined in detail.

The syntax and semantics of AvDL inherit much from the C [Kerninghan and Ritchie 1988] family of programming languages and are largely based on those of the C++ programming language as described by Stroustrup [1997]. As such, AvDL program source code largely resembles programs written in the C/C++ family of languages. It has to be noted, however, that some fundamental elements of the core language (detailed below in Section 8.2.1) are different from their counterparts in C/C++:

- The language only includes a single numeric data type, the scalar data type.
- The language has three different kinds of array types.
- There are a number of additional game AI related data types.
- AvDL does not use unions and enumerated data types.
- In AvDL pointers are not used and addresses cannot be accessed, i.e. there is no direct memory access mechanism.
- Classes provide the only means to create record structures in AvDL, i.e. there are no 'struct' records.
- There is no data hiding in AvDL classes.
- In AvDL there is no multiple inheritance of classes.
- The language does not support function inlining.
- All function parameters in AvDL are passed by reference unless specified otherwise.
- The AvDL specification adds several (program flow) control structures to the common C/C++ control structures.

The AvDL system provides mechanisms that allow NPC behaviour definition through the creation of an annotated world (see Chapter 2, Section 2.3.4.4). This feature is described in detail in the discussion of the SEAL (Simple Entity Annotation Language) subset of AvDL which uses entity annotation as its main method of behaviour definition (see Chapter 9).

Despite the danger of leading to confusion, AvDL retains most of the more abstract elements of C/C++, such as C/C++-style iterations and selection, which may not be easy to understand by novice programmers. Some additional macro-like syntactic synonyms (aliases) which can be substituted either by a preprocessor or by the compiler itself are integrated into the language specification as alternative means for expressing NPC programs. This should present novice programmers with several simpler and easier to read alternative notations without

the language having to sacrifice its similarity to its C/C++-like syntax. All variable data types in AvDL are auto-initialising. This means that a variable that is declared will initially be provided with a default value unless otherwise specified by the AvDL program. This not only simplifies the declaration process but also removes one of the major shortfalls of common implementation languages as errors caused by uninitialised values are often hard to identify and debug. The presence of auto-initialisation helps to avoid this type of error within AvDL programs. Furthermore, inspired by the C/C++ programming languages, variables in AvDL are strongly typed, a feature in which the language differs from many other scripting languages.

AvDL handles object orientation in a manner that is similar to C++, however, there are a number of notable differences, especially the introduction of implicit classes (see Section 8.2.1.1). In addition to its handling of object orientation, AvDL also differs from languages such as C and C++ through its data types. The definition of AvDL includes a number of specialised data types (see below) that do not exist in the definition of the C/C++ family of programming languages. Furthermore, in addition to these data types, AvDL also provides several control structures that have no equivalent in the C/C++ family of languages but which have proven useful in other programming languages, such as a conditional alternative that allows the specification of an additional separate condition ('elsif').

8.2.1 The Syntax of AvDL

AvDL programs are meant to encode complete NPCs and as such each program needs to be declared as an 'entity' (see Figure 8.1) in which the NPC's program is encapsulated (in a similar manner to C++ namespaces). The identifier used to name the entity is then used to mark the AvDL program's entry point from which execution will commence using a syntactical notation inspired by the C++ class constructor. This provides a more consistent approach to defining the program entry point than found in C/C++ which use a function called "main", as here there is no need to use reserved identifiers. Apart from this, as mentioned before, the syntax of AvDL is very similar to the syntax of the C++ programming

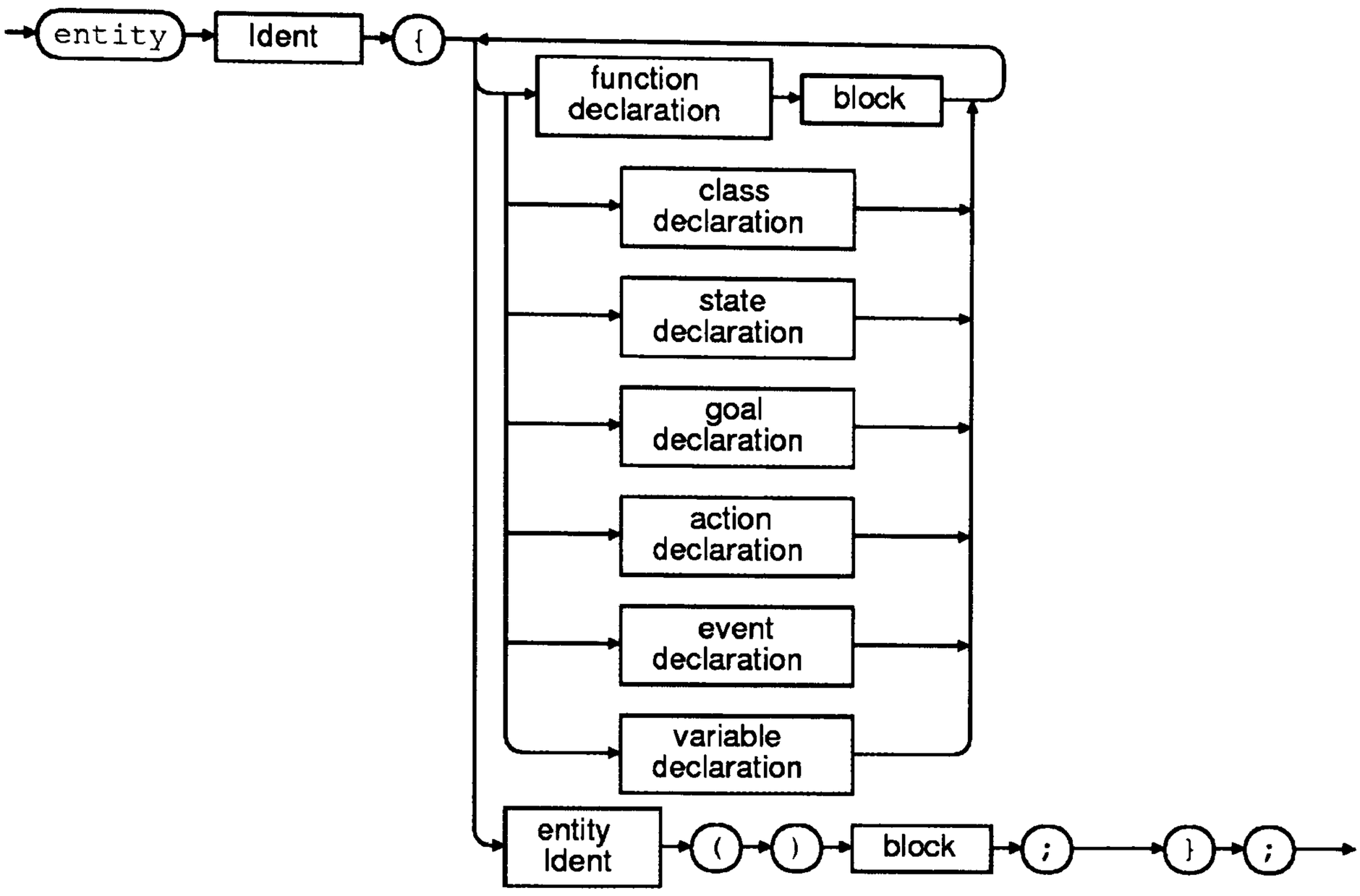


Figure 8.1: Syntax for declaring an 'entity' object.

language. This influence of the C++ programming language on AvDL is not only visible in AvDL's implementation of classes and inheritance (see Section 8.2.1.1), but is also evident in the definition of functions in AvDL programs which is very similar to that of functions written in the ANSI C++ programming language. As is the case in C++, the forward declaration of functions in AvDL also uses prototypes in the form of function heads. A function has a name (its identifier), a return data type and a list of formal parameters. If the return data type is omitted during function declaration (or definition), the AvDL compiler will assume that the return data type is the typeless 'void'. Similarly, the omission of parameters in a function declaration will default to an empty parameter list presumption which is equivalent to declaring a 'void' parameter list.

Scoping in AvDL, defining where within a program's source code it is legal for the program to access data or to call a function and where these are visible, is

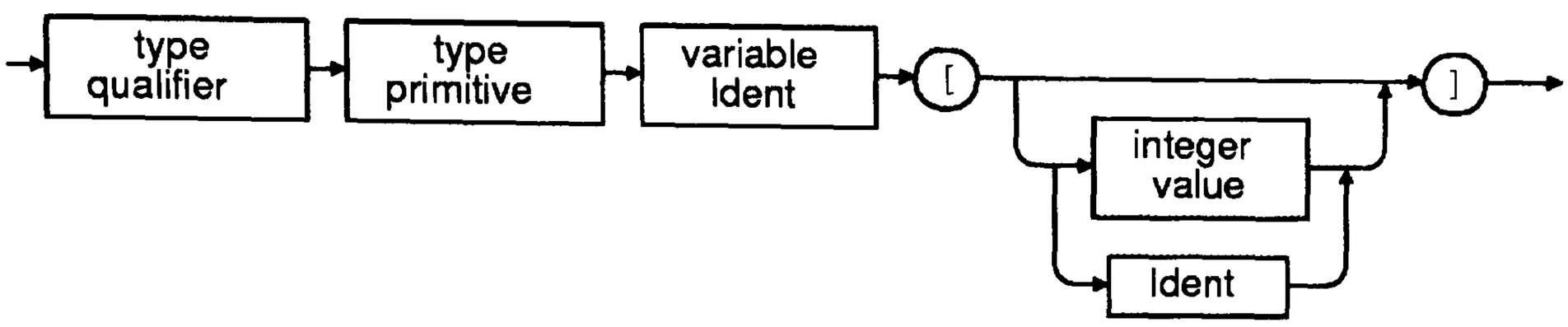


Figure 8.2: Declaration and use of arrays in AvDL.

handled in a manner that is very similar to C/C++, i.e. constants and variables that are defined globally can be accessed from anywhere in an AvDL program, while constants and variables that are defined locally within a block can only be accessed from inside this block. The scope of a function in AvDL extends from directly below the place of the function declaration within the current module (NPC program source code file) until the end of the file. While in the C programming language it is possible to declare a function locally by placing its prototype within the body (the definition) of another function, it is not possible to do this in AvDL. The reason for this restriction is that the (prototype) declaration of a function within a different function, which would be disjunct from its definition, as this would have to be located below the function it was declared in, would break not only the consistency, but also the simplicity design principles for BDLs (see Chapter 5, Section 5.3.1). The AvDL specification also does not allow the nesting of function definitions as would be possible in programming languages related to Pascal, such as ZBL/0 (see Chapter 7, Section 7.2).

There is only one non-specific numeric data type in the AvDL that can take floating point values as well as integer values, which is the 'scalar' data type. This means that there is no differentiation in the way that AvDL handles the 'short', 'int', 'long', 'float' or 'double' data types found in C/C++. Instead, a number of aliases, implemented as synonyms in the preprocessor of the AvDL compiler, will allow for variables to be declared using the familiar C/C++ numerical data types, thus eliminating the need for a specific casting operator for type conversions.

Nevertheless, variables in AvDL are strongly-typed as type-safety is important to prevent programming errors when it comes to user-defined functions with formal parameters.

Unless explicitly initialised to a different value, the AvDL virtual machine auto-initialisation will default all numeric variables to the value '0' (zero) while some of the game AI specific data types (discussed below) are auto-initialised to the value 'NULL'.

There is a separate data type for Boolean values only, the 'bool' data type. Variables of this type can take the values 'true', implemented as a preprocessor alias mapped to the value '1', or 'false', a preprocessor alias mapped to the value '0'.

Variables of the 'scalar' data type may also hold Boolean values which can be achieved if the data held in a scalar variable is assigned to a Boolean variable. In this case the data is automatically downcast and all non-zero values are interpreted as 'true'. Boolean variables are auto-initialised to the value 'false' by the AvDL virtual machine.

AvDL programs allow for groups of variables of the same type to be stored in variables of an aggregate data type. For this AvDL provides three different kinds of arrays (see Figure 8.2): static (fixed-size) arrays, dynamic arrays and associative arrays.

Some of the differences between these data structures are transparent to the user, i.e. syntactically there is little difference in their usage, as access to array elements is always granted by using the subscript operator. The usage of these data structures manly differs in the declaration of the data structures and in the method by which array elements are addressed, i.e. through a numerical index (starting at index '0' for the first element as is the case in the C/C++ programming languages) or an associative value:

- Static arrays in AvDL are variables of any AvDL data type that have been declared using the [] subscript operator with a size (of array) indicator as their suffix. As such they are almost identical to arrays in C, however, unless the elements of an AvDL static array are manually initialised during the array's declaration, they will be auto-initialised to the value '0' (zero).

- Dynamic arrays in AvDL are variables of any AvDL data type that have been declared using the [] subscript operator without a size (of array) indicator which are subsequently given a size using the 'new' operator. After its declaration, a dynamic array is by default pre-initialised to the empty value 'NULL'. The memory that has been allocated for dynamic arrays using the 'new' operator must be released eventually, using the 'delete' operator. Unlike in C++ the 'delete' operator in AvDL does not need to be succeeded by the subscript [] symbol as a requirement for freeing memory that has been allocated to arrays of data. The use of the 'new' and 'delete' operators in AvDL is restricted to dynamic arrays.

- Associative arrays, inspired by Perl [Schwartz 1992], are variables of any AvDL data type that have been declared using the [] subscript operator without a size (of array) indicator. Associations are created dynamically as soon as they are used for the first time. Internally each new association is given an increasing index value.

The specification of AvDL does not include pointers, i.e. it is not possible to directly access memory blocks within the address space of AvDL programs or the addresses to data held in the memory of the AvDL virtual machine. There are several reasons for this omission of pointers, first among which is the observation that the understanding of pointers (or rather a lack thereof) frequently provides one of the main stumbling blocks for novice programmers and therefore would unnecessarily complicate the language. Another reason, which is possibly more important, stems from the system's mechanism for object annotation (see Chapters 9 and 10). This allows access to program segments of entities controlled by AvDL programs other than the currently executing one, and if direct memory access were allowed, its effects could seriously disrupt program flow and destabilise the run-time environment. Consequently, and thus unlike the C or C++ programming languages, AvDL provides no mechanisms for pointer arithmetic or access to data held in arrays without the use of the subscript operator.

AvDL retains the (program flow) control structures of the C/C++ family of languages, i.e. all of the familiar iterations and selections work in an identical manner to their C/C++ equivalents. The conditional 'if', 'if'-'else' and 'switch'

statements are not the only selections available in AvDL. These control structures are complemented by an additional type of conditional alternative that allows the specification of a second, separate condition ('if'-'elsif'-'else' and 'if'-'elsif'), as well as two further multiple alternatives, 'select' statements, the first of which is almost identical to 'switch' statements except for the fact that its cases do not contain fall-troughs. The second 'select' statement allows the selection of not only single alternatives but also ranges of values to identify the statement that is to be executed. In addition to the 'while', 'do'-'while' and 'for' loops, the AvDL specification also implements three further iterations, i.e. a foot controlled 'repeat'-'until' loop (as found in Pascal), a 'do'-'forever' continuous loop and a looping control structure for accessing array elements ('foreach'-'of') that cycles through arrays (static, dynamic and associative), allowing each array element to be processed in turn.

8.2.1.1 Object Orientation in AvDL

Object orientation (OO) in AvDL resembles OO in C++ and Java with some features being closer to Java than C++ and several features different to both C++ and Java. The data structure that allows OO in AvDL is the 'class' compound data structure which is identical to the above two languages. Classes in AvDL are used to describe objects (in the sense of object orientation) as well as record structures (in the ANSI C 'struct' sense). AvDL's equivalent to the top-level class found in Java programs is the 'entity' object (see Section 8.2.1).

Unlike classes in C++, classes in AvDL do not support mechanisms of data hiding that would restrict access to their attributes (data members) and methods (member functions). This means that all methods and all attributes of a class are public (in the C++ sense). There is no equivalent to protected or private class components as found in C++. Within the scope of an instance of an AvDL class, the AvDL program has full access to all attributes and methods defined by the class. By default all classes in AvDL have one implicit attribute – a reference to the current instance of the class – that can be accessed through the 'this' object. The 'this' object reference is also a hidden parameter which is implicitly passed to all methods of the class as the first parameter of the method.

There is no function inlining in AvDL, i.e. unlike in the C++ programming language there is no support for inline functions in AvDL. The main implication of this is that although the methods of a class are declared within the class description, the methods themselves must be defined below the class definition itself. The decision to omit function inlining was made for reasons of language simplicity, i.e. to avoid confusion through allowing too many different methods for defining an object's methods and to impose a strict distinction between class declaration and definition of its functionality.

AvDL implements the concept of implicit class definitions that allows for class definitions to be stored within external files. We believe this will benefit AvDL program modularity and encourage the use of parallel development.

The file containing an implicit class definition must either be a valid AvDL source code file containing only the class definition (any other code will be ignored) in source code form or alternatively a pre-compiled class definition (similar to pre-compiled classes in the Java programming language) as bytecode for the virtual machine. The name of this file must be stated at the declaration of the implicit class. Like Java, AvDL does not support multiple inheritance. As a consequence of this an object's class in AvDL can only be derived from a single class using Java's 'extends' statement (also allowing the ": public" notation from C++ as an alternative). Currently there are no plans to allow inheritance from implicitly defined classes that have been pre-compiled.

The current specification of AvDL does not support function overloading or function overriding. While these features provide very powerful mechanisms in the languages that include them, we believe that they would be beyond the scope of AvDL which is only supposed to provide a behaviour definition extension language to computer game applications, adding an additional layer of complexity that would outweigh any benefits gained by the inclusion of these features in the language specification.

8.2.1.2 Triggers and Event Based Programming in AvDL

The language specification includes an 'event' data type (see Figure 8.3) to allow NPC programs to react to named events (using the given identifier) i.e. AvDL

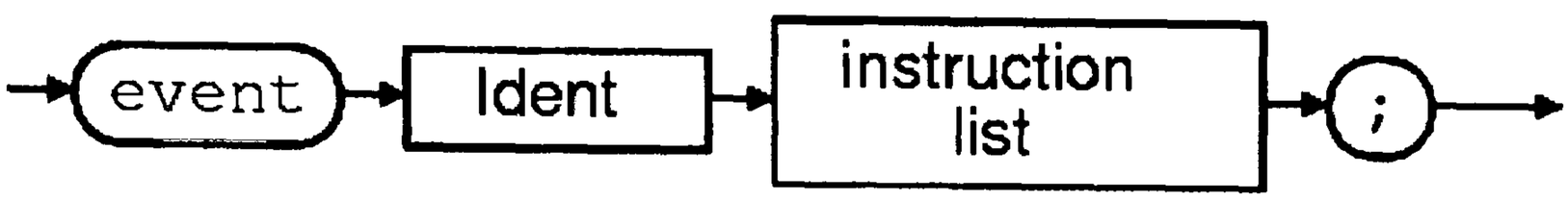


Figure 8.3: Syntax for declaring an event with event-handler (instruction list).

is effectively useable as a Trigger-Only Induced Script type ($ST2$) scripting language. if the situation demands this. For events defined in the host application the event registration is exposed (made accessible) through the run-time API (see Chapter 10, Section 10.4). The declaration of the 'event' requires the definition of an AvDL instruction list, i.e. an event handler that will be triggered once an event occurs.

In addition to the event handler, a second mechanism to enable NPC programs to react to events exists in the form of scalar and Boolean AvDL variables that have been declared using the 'triggered' type qualifier for a given event which will be set to the value '1' or 'true' when that event occurs.

Finally, the current specification of the language introduces a 'trigger' operator for spawning events from within NPC programs. Events that have been triggered this way are spawned globally throughout the run-time environment unless they are addressed directly towards a specific entity. The operator returns a Boolean value to report success ('true') or failure ('false') of triggering the event. This functionality could have been exposed through a function from a standard library or alternatively through a special statement for program flow control, similar to

statements such as 'break' or 'return', but the use of an operator for this purpose is much more consistent with the structure and makeup of the AvDL.

8.2.1.3 A Data Type for State Machines

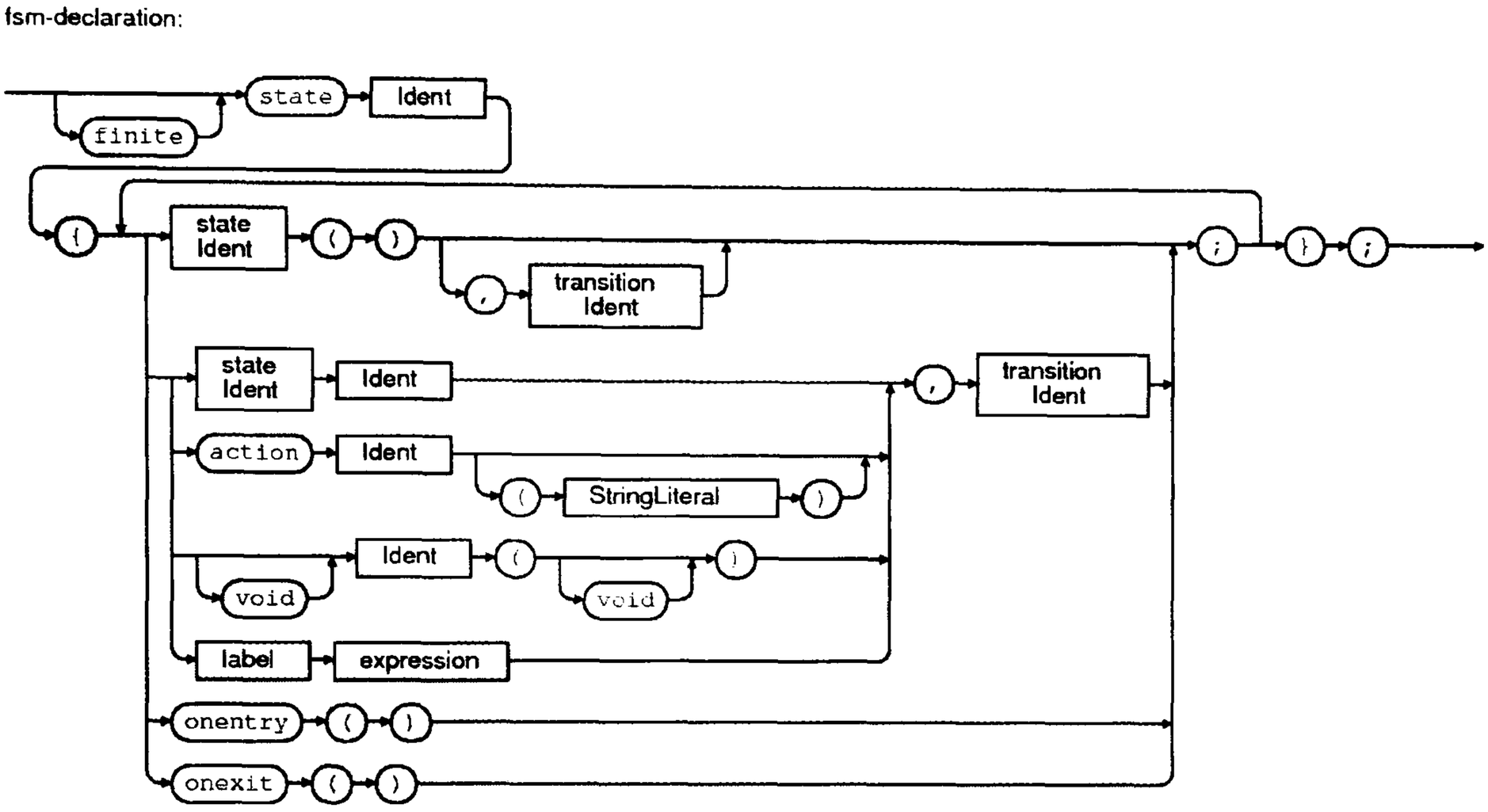


Figure 8.4: Syntax for FSM declaration.

As we have already stated (see Chapter 2, Section 2.3.1.1), FSMs provide a tried and tested mechanism which has proved suitable for many kinds of computer game AI which makes them by far the most used AI technology in modern computer games [Anderson 2003a] as they allow for the simple definition of deterministic behaviour. For this the AvDL specification provides a 'state' data type that allows the definition of state machines (finite as well as fuzzy) which can also be used to express the structure of hierarchical state machines [Fu and Houlette 2004], i.e. each state can also be a complete state machine. Each state can only have a single instance which is automatically created when a state is declared, i.e. any variables that are declared of a state are references to this state instance. Of these two 'state' types the finite state machine type (the default state machine type) is included in the specification of the SEAL subset of the AvDL scripting

language (see Chapter 9, Section 9.3). The implementation of this type within the virtual machine of the AvDL system is described in detail later in this thesis (see Chapter 10, Section 10.2.3).

8.2.1.3.1 Finite State Machines in AvDL The 'state' data structure defining a state machine bears some similarities to the 'union' data type found in the C programming language, as at any one time only one state within it will be fully active. It also shares elements with the definition of a 'class' data structure, as its members are declared within the structure similar to an object's methods and defined outside of the structure itself. It is the default 'state' data type in AvDL, so it can be declared with or without the presence of the 'finite' type qualifier. Members of the state structure can be used as identifiers for states in a similar manner to the named constants of C enumerated data types (also addressable as members of their parent state structure in case of name conflicts). Each state within a state construct, i.e. each state structure member, needs to be provided with a follow-up (next) state to declare which state the current state will change into during an automatic state transition (see Figure 8.4), i.e. a state transition that occurs when all of the instructions for the current state have been executed. If the value 'NULL' is used as a transition target, the state machine will terminate automatically when this transition occurs.

These state structure members can be of different types:

1. A reference to an instance of another state machine structure (referenced through its identifier), effectively providing an alias for addressing that state and allowing that state's transition target to be overridden.

2. An action (see Section 8.2.1.5), i.e. a function in the host application, which in this case should not return any data and be parameterless, as this would be ignored by the FSM.

3. An AvDL function (similar to a method in a class), which must be a typeless function with an empty parameter list.

4. A labelled AvDL expression, addressable through the label.

Independent of their actual types, these structure members will always be treated as states by the FSM.

Inspired by the syntax for the 'entity' object's entry point as well as the C++ constructor, the instructions associated with the state of the 'state' data type itself need to be placed within a special method (member function) of the state structure, the state's body, marked by the identifier used to name the state structure. The declaration of this method includes the state's transition target (follow-up state), which defaults to the value 'NULL' if this transition target is omitted. The 'state' structure in AvDL allows for the definition of two further specialised methods, 'onentry' and 'onexit', the former of which will be executed before the state's body is entered, whereas the latter is executed when a state's body is exited due to a state transition. These two functions are not unlike the constructor (method which is invoked when an object is created) and destructor (method which is executed when an object is destroyed) of an object oriented class. If these functions are not explicitly declared within a state structure and defined among the state's members, default methods (defined within the AvDL run-time system) will automatically be used instead.

Until an initial state within a state machine has been set, it will hold the empty value 'NULL', so before a state machine starts its execution it needs to be initialised and set to an initial state. This is done using the unary 'setstate' operator which returns a reference to the state instance that is set or to the instance of the parent state structure if the set state is not a state data structure. The 'setstate' operator can be used to set any state or state member to be the currently active state. In case of name conflicts, state members can be addressed as members of their parent state. The status of an FSM structure or structure member can be queried and is always a Boolean value, showing if a state is currently active ('true') or inactive ('false').

Once the initial state has been set, the state machine will start its execution, diverting program flow to the state machine until it terminates, i.e. until it takes on the empty value 'NULL'. While a state machine is running, the 'setstate' operator can be employed from within the state machine to force a transition to any state (or state member) that has been declared in the program or to terminate the state machine by using 'NULL' as its operand. Finally, the currently set

state can be queried using the 'getstate' operator which returns a reference to the currently active state (or state member), or 'NULL' if no state machine is active.

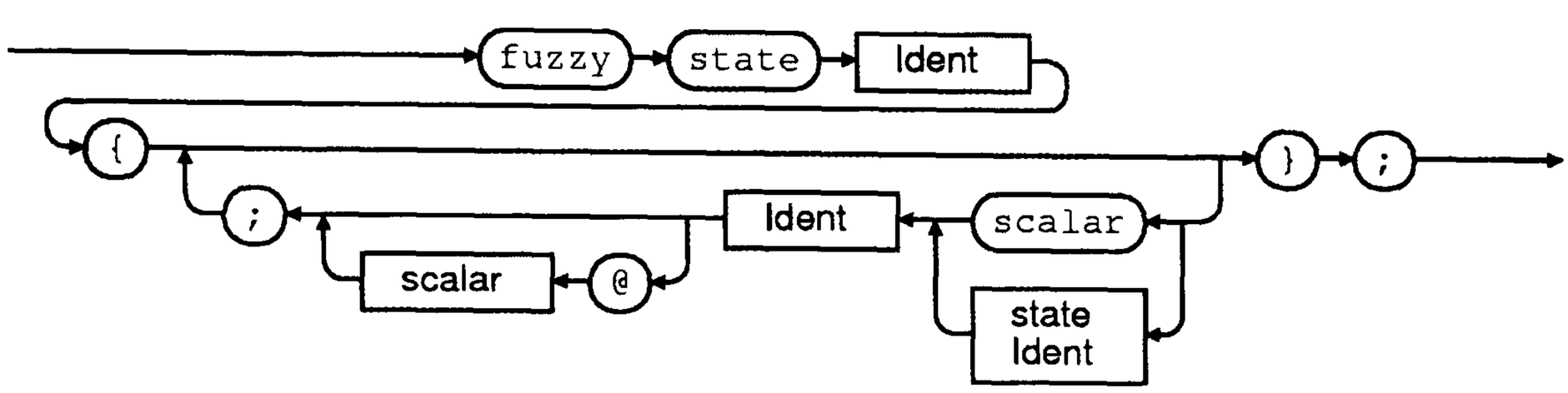


Figure 8.5: Syntax for FuSM declaration.

8.2.1.3.2 Fuzzy State Machines in AvDL The finite state machine type was one the first special types to be included in the AvDL specification and its makeup and functionality have changed little since this first specification, whereas in the case of the fuzzy state machine (FuSM) type the data type described here is still untried and tentative, however, syntactically it is the most likely candidate for inclusion in the system (see Figure 8.5).

Structures of the FuSM 'state' data type in AvDL are declared with the 'fuzzy' type qualifier. For the declaration of state structure members for a fuzzy state machine, instead of a transition target an optional weight value for the (member) state (capped between the scalar values '0.0', the default value for weight declarations, and '1.0') can be provided. They, too, can be used as identifiers for states in a similar manner to the named constants of C enumerated data types, but the data types of members in FuSMs are restricted to being a reference to an instance of another fuzzy state machine structure (referenced through its identifier), or a data member (variable) of the scalar data type (capped between the values '0.0' and '1.0'), i.e. FuSM structures in AvDL do not have member functions, such as the methods in the FSM data structure.

Until an initial state within a fuzzy state machine has been set, just like its FSM equivalent it will hold the empty value 'NULL'. An FuSM is activated using

the unary 'setstate' operator which returns a reference to the state instance, and which in case of a fuzzy state structure member optionally allows the specification of a weight value for the state (capped between the scalar values '0.0' and '1.0', the default value in 'setstate' operations). Here, too, the 'setstate' operator will return a reference to the instance of the parent state structure if the set state is not a state data structure.

The status of an FuSM structure or structure member can be queried and is always a scalar value, showing the degree of activity of the state. If queried, by default a fuzzy state structure will return the accumulated value of its members as a scalar value (that may be larger than the value '1.0'). This is also true if the queried state structure is a member of another structure, however, it will return the value that it has been set to in its parent structure if it is addressed through this parent state structure, as it may be a member of several state structures in each of which it may have been given a different fuzzy weight value.

The 'setstate' operator can be used to modify the value of a state member's weight relative to its existing value by augmenting the specification of the weight with a sign. If this is the case, the modified weight of the state member will be automatically capped between the scalar values '0.0' and '1.0', i.e. no state member can grow or shrink to a weight value smaller than '0.0' or larger than '1.0'. For FuSMs the AvDL specification provides no equivalent to the 'getstate' operator used with FSMs, as more than one state can be active at any one time.

This shows that, while having a large degree of syntactic similarity with the FSM type, the FuSM type works entirely different from the FSM, i.e. FuSMs exist as a sort of record data structure but they do not hold code that executes. The main difficulty when it comes to the definition of an FuSM is that in game development there appears to be little consensus as to what a fuzzy state machine is and how it works with definitions varying between the formal computer science understanding of the term to methods involving probability and random selection between states [Champandard 2004]. The semantics for the FuSM data type have not been finalised in the current specification of the language, however, the semantics of the different FuSM definitions are all expressible through the FuSM syntax described here. These different possible approaches are further discussed later in this thesis (see Chapter 11, Section 11.2.3).

8.2.1.4 A Data Type for Goal-Orientation

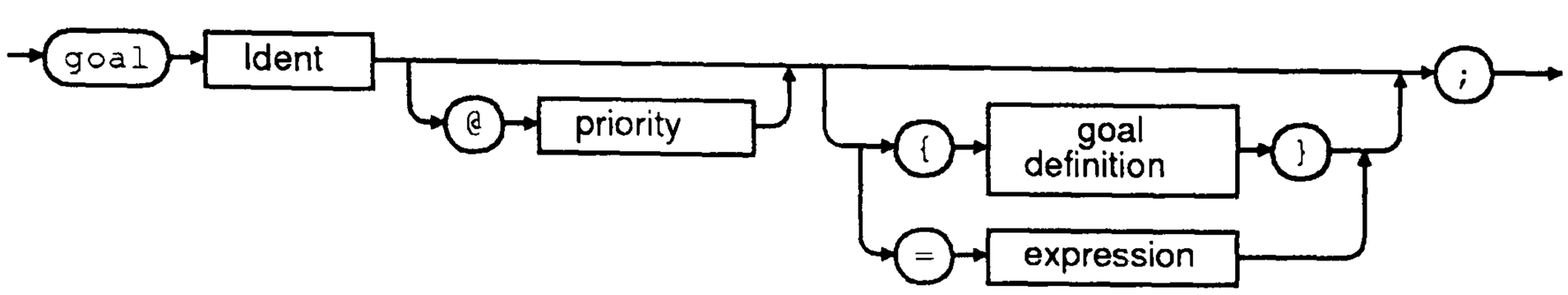


Figure 8.6: Syntax for declaring a goal.

Goal-directed behaviour is one of the simplest forms of nondeterministic behaviour. A goal is the end-state of a set of goal-directed actions. The path that the system needs to take to reach this end-state is generated by using a planning heuristic on a set of known values which need to be conveyed to the AI module beforehand. The generation of this sequence of actions that will lead to the desired goal is called goal-oriented action planning [Orkin 2004a] (see also Chapter 2, Section 2.3.2.2).

In AvDL goal-directed behaviour can be attained through the use of the language's 'goal' data type (see Figure 8.6). Unlike the 'state' data type, this data structure does not form a part of the SEAL subset of AvDL, i.e. the current system prototype does not yet incorporate all of the data structures that goal orientation would require for it to be fully implemented for use in the system's run-time environment (see Chapter 10, Section 10.3). The makeup and functionality of this 'goal' data type has been inspired by the planning mechanisms of the behaviour definition language CML [Funge 1999] (see also Chapter 5, Section 5.1.1.3), as well as Orkin's description of GOAP [Orkin 2004a].

The combined set of goals that have been declared within an NPC program make up the search space from which action plans can be generated. An instance of a goal data structure holds implicit data members which contain an action plan and a reference to the current state of the goal (in relation to that action plan). Until an action plan for a goal exists, the goal state will have the queryable value 'NULL'. Once an appropriate action plan has been generated, the goal's status will be set to the Boolean value 'false' until the goal state has been reached, resulting in the goal's status being set to the Boolean value 'true'.

In its simplest form, a goal is defined as a single variable of the 'goal' data type, optionally at a set priority or weight for the planner (by default set to '1.0'), which has been assigned an expression defining the exact preconditions that have to be satisfied for the goal itself to be reached. Priorities (weights) are given higher values as their importance shrinks, e.g. a value of '1.0' has a higher priority than a value of '3.5'. The reason for assigning lower values to higher priorities is that informed search methods, which are commonly employed to generate plans in goal-oriented systems, "typically use some estimated measure ... and try to minimize it" [Russel and Norvig 1995].

The second method of using the 'goal' type requires the declaration of the goal as a compound data structure with optional priority (default value '1.0'), simplifying the declaration of goals with several preconditions. Each of the structure's members is a precondition (or sub-goal) which needs to be satisfied for the goal to be reached, allowing AvDL programs to use composite tasks as described by Dybsand [2004].

All 'goal' structure members can be used as identifiers in a similar manner to the named constants created in C enumerated data types. They take the form of labelled AvDL expressions which can optionally be supplied with a priority or weight (by default set to '1.0') which will be taken into account by the planner and which must evaluate as true for the goal to be reached.

An alternative interpretation of the goal structure would have been to use priorities to convey preference knowledge during planning, providing a weighting only to sub-goals contained within the structure instead of the goal itself, with only the reachable condition with the highest possible priority required to be satisfied while its sibling goals (preconditions) would have been considered optional.

However, the employment of this alternative goal structure would unnecessarily complicate the plan generation in the run-time environment, e.g. it could prevent the use of a standard planning algorithm, such as A*, for the planning process.

The AvDL specification defines a number of specialized operators for goal-oriented action planning. The operator for the creation of the action plan itself is the 'plan' operator which directly operates on goals, generating a plan from all valid goals in the NPC program. The syntax of this unary operator is similar to the 'new' operator in the C++ and Java programming languages, and as such also similar to the 'new' operator of AvDL. If the generation of a plan has been successful, the operator will return an instance of the goal as its result. If the planner is unable to find a suitable plan for this goal, the empty value 'NULL' is returned. By default the plan operator in the AvDL virtual machine will be expected to use A* planning (see Chapter 4, Section 4.3.3) as the underlying planner, thus implementing the GOAP technique suggested by Orkin [2006]. To allow for greater customisation, the run-time API will provide a planner interface to enable different planning methods to be implemented by the NPC program developer.

As soon as a plan for a goal has been generated the goal variable's status, as well as the status for all of the nodes in the plan's action sequence will be set to hold the Boolean value 'false' and the plan will start executing. The expressions that define a goal's preconditions may contain any function calls or actions (see Section 8.2.1.5) that may need to be executed to meet the precondition. These will automatically be evaluated while the action plan executes. The unary operator 'reached' can then be used for testing a goal for completion (i.e. testing the goal state). This operator is required for querying the status of a goal's plan, as strong typing of data types prevents direct access to variables of the goal data type. While the action plan is still being executed, the 'reached' operator will continue to return the value 'false'. If a goal has been reached (by all preconditions or sub-goals having been fulfilled), its status will change and the 'reached' operator will return the Boolean value 'true' as its result. If the situation in the NPC's virtual environment changes in a way that an action plan becomes invalid, i.e. if it is no longer possible for the NPC to reach its goal, the goals status will change

and the 'reached' operator will result in the value 'NULL', usually requiring a new plan to be generated.

8.2.1.5 Mechanisms for Accessing the Host Application from AvDL

NPC programs can be enabled to directly call functions that are defined within the run-time environment's host application. These function bindings are created by using the AvDL data type 'action', not to be confused with actions in GOAP (see Section 8.2.1.4), which maps AvDL actions to functions in the host application. Functions that are mapped to actions within AvDL programs need to be declared to the AvDL virtual machine by the host application through API calls to the run-time environment (see Chapter 10, Section 10.4). By default, functions that have been registered in this way are then bound to the action whose identifier corresponds with the name of the function. Alternatively the name of the function can be associated with the action through a string which is given as a parameter. If the mapped function in the host application expects parameters, these can be declared similarly to the declaration of formal parameters in a function prototype.

Actions are auto-initialised to the value 'NULL' which they will hold until the first time the action is executed. Any data returned by the function in the host application is stored within the action data type for retrieval in the NPC program. If the function in the host program does not return any data to the action it will default to the value 'true'. Actions that are needed within an AvDL object only (see Section 8.2.1.1) must be declared as members of that object. Actions that have been declared in an AvDL object are bound to the corresponding functions as soon as the constructor for that object is called during program execution. All of these function bindings are released when the destructor for the last instance of this class is called during program execution. Actions that have been declared in an FSM structure are bound to the corresponding functions as soon as the state machine is initialised for the first time. These function bindings are released when the NPC program terminates.

Actions that execute functions in the host application are not the only mechanism by which NPC programs can interact with their host. Data in the host application can be bound to scalar and Boolean AvDL variables in NPC programs

by using the 'volatile' variable attribute. The 'volatile' attribute is used in the C and C++ programming languages to mark data that is influenced by processes which are external to the current program. Similarly by making a variable in an AvDL program 'volatile' it can be mapped to a variable in the host application through the API of the run-time environment by using the identifier given to the variable when it was registered with the API.

8.3 Using AvDL to Create NPCs

The use of AvDL for defining the behaviour of a virtual entity is quite straightforward. The NPC as a whole is encapsulated within an 'entity' object that explicitly provides an entry point for code execution, effectively the main NPC program. This entity object contains the various data structures that define the behaviour of the virtual entity.

This can be through the use of popular game AI methods, such as the definition of finite state machines that will control the NPC, or the use of a trigger system that defines a reactive, event based NPC. Alternatively, newer techniques, such as GOAP can be used to create an NPC with nondeterministic behaviour.

8.3.1 An AvDL FSM Example

The earlier example for a typical FSM in games (see Chapter 4, Section 4.2.1) described an NPC on patrol, carrying out guard duty. The FSM in the example consisted of the states 'patrolling', 'challenging intruder' and 'attacking intruder' (see Figure 4.2). Assuming that the sensor inputs for that program were implemented through sensor variables of the NPC in the host application that are mapped through the 'volatile' type qualifier, the structure of a possible version of this program would look as follows in AvDL:

```
entity guard
{
    volatile bool intruder_detected;
    volatile bool intruder_hostile;
```

```
volatile bool intruder_friendly;
volatile bool intruder_dead;

state patrolling
{
    patrolling(), challenging_intruder;
};

patrolling::patrolling()
{
    do
    {
        if(intruder_detected)
            setstate challenging_intruder;
        /* execute 'patrolling' behaviour */
        ...
    } forever;
}

state challenging_intruder
{
    challengine_intruder(), patrolling();
};

challenging_intruder::challenging_intruder
{
    while(!intruder_friendly)
    {
        if(intruder_hostile)
            setstate attacking_intruder;
        /* execute 'challenging_intruder' behaviour */
        ...
}
```

```
}

    state attacking_intruder
    {
        attacking_intruder(), patrolling;
    };

attacking_intruder::attacking_intruder
{
    while(!intruder_dead)
    {
        /* execute 'attack intruder' behaviour */
        ...
    }
}

state fsm
{
    fsm();
};

fsm::fsm()
{
    setstate patrolling;
}

guard()
{
    setstate fsm;
}
};
```

In this example, each state, including the FSM itself, is represented by its own state data structure. The makeup of the state structure in AvDL allows an alternative expression of the same NPC program in which all states are stored within the same state machine structure:

```
entity guard
{
    volatile bool intruder_detected;
    volatile bool intruder_hostile;
    volatile bool intruder_friendly;
    volatile bool intruder_dead;

    state fsm
    {
        patrolling(), challenging_intruder;
        challenging_intruder(), patrolling();
        attacking_intruder(), patrolling;
        fsm(), NULL;
};

fsm::patrolling()
{
    do
    {
        if(intruder_detected)
            setstate challenging_intruder;
        /* execute 'patrolling' behaviour */
        ...
    } forever;
}

fsm::challenging_intruder
{
```

```
while(!intruder_friendly)
{
    if(intruder_hostile)
        setstate attacking_intruder;
    /* execute 'challenging_intruder' behaviour */
        ...
    }
}

fsm::attacking_intruder
{
    while(!intruder_dead)
    {
        /* execute 'attack intruder' behaviour */
        ...
    }
}

    fsm::fsm()
    {
        setstate patrolling;
    }

    guard()
    {
        setstate fsm;
    }
};
```

8.3.2 An AvDL Trigger System Example

If AvDL is used as a Trigger-Only Induced Script type ($ST2$) scripting language using an event based programming style that same scenario could be represented as shown below:

```
entity guard
{
    scalar behaviour = 0;

    event intruder_detected { behaviour = 1; };
    event intruder_hostile { behaviour = 2; };
    event intruder_friendly { behaviour = 0; };
    event intruder_dead { behaviour = 0; };

    guard()
    {
    do
    {
        select(behaviour)
        {
            case 0:
            /* execute 'patrolling' behaviour */
            ...
            case 1:
            /* execute 'challenging_intruder' behaviour */
            ...
            case 2:
            /* execute 'attack intruder' behaviour */
            ...
            }
        } forever;
    }
```

```
};
```

8.3.3 A Nondeterministic NPC Example

A similar program can be expressed using GOAP. Again, assuming that the NPC's sensor inputs are variables that have been mapped through the 'volatile' type qualifier, a nondeterministic solution for this scenario could look as follows:

```
entity guard
{
    volatile bool intruder_detected;
    volatile bool intruder_hostile;
    volatile bool intruder_friendly;
    volatile bool intruder_dead;

    bool attack_intruder()
    {
        while(!intruder_dead)
        {
            /* execute 'attack intruder' behaviour */
            ...
        }
        return true
    }

    goal enemy_killed = attack_intruder();

    bool challenge()
    {
        enemy_killed defended;

        while(!intruder_friendly)
```

```
{
    if(intruder_hostile)
    {
        while(reached(defended)==NULL)
        {
            defended = plan enemy_killed;
        }
        return true;
    }
    /* execute 'challenging_intruder' behaviour */
    ...
}
return true;
}

goal handle_intruder = challenge();

goal protect = reached(handle_intruder);

guard()
{
    protect defend_objective;

    do
    {
        if(intruder_detected)
        {
            while(reached(defend_objective)==NULL)
            {
                defend_objective = plan protect;
            }
        }
    }
    /* execute 'patrolling' behaviour */
```

```
            ...
        } forever;
    }
};
```

Chapter 9

The Simple Entity Annotation Language

SEAL, the Simple Entity Annotation Language [Anderson 2005b] is part of the AvDL system (see Chapter 8) for the definition of believably intelligent game character behaviour. As a BDL coupled with the concept of "Smart Terrain", as described by Forbus and Wright [2001], SEAL presents a promising combination of NPC behaviour definition techniques. Combining rule-based systems with affordance theory, the embeddable Regular Script type (*ST3b*) scripting language SEAL provides a unified approach to the definition of virtual entities within one behaviour definition language for virtual entities as well as the "smart" objects that the entities can interact with.

9.1 SEAL within AvDL

SEAL is a 100% compatible subset of the AvDL scripting language, effectively making SEAL a module of the AvDL system. This means that SEAL programs are source code compatible with AvDL, i.e. valid SEAL source code is automatically valid AvDL source code and should compile on an AvDL compiler.

The language's syntax is defined as a reduced version of the AvDL syntax, also using an LL(1) grammar (see appendix E). The language is kept much simpler than AvDL through the omission of some of the more complex features of AvDL

(see Section 9.3 below), but remains sufficient for the creation of rich virtual environments, populated by virtual entities that interact with the game world.

If an NPC program that encodes a virtual entity in an annotated world requires the use of features that are not incorporated within SEAL, the more complex AvDL should be used to express the NPC program.

9.2 Entity Annotation for NPC Behaviour Definition

SEAL is a BDL that is dedicated to the creation of NPCs that inhabit an annotated world. The mechanism for creating annotated worlds (see Chapter 2, Section 2.3.4.4) that we refer to using the term "Annotated Entities" has been described using various names, such as "Smart Terrain" [Cass 2002], "Smart Objects" [Peters et al. 2003; Orkin 2006] and "Annotated Environment" [Doyle 2002], all of which are generally interchangeable and mostly used with very similar meanings, although slight differences in their exact interpretation sometimes remain. A common aspect to all of the implementations that utilise this mechanism is the indirect approach to the creation of believable intelligent entities. Such intelligent entities that inhabit the virtual world do not have the knowledge that would enable them to interact with other objects of the world that can be interacted with, but these objects themselves have the knowledge as to how other virtual entities can interact with them. These objects broadcast information about themselves (including the instructions on how to use them) to NPCs in their vicinity, which can then use this information for interaction with those objects, making the objects "smart". NPCs only passively interact with objects, meaning that effectively the objects interact with themselves through the mediation of the entities that appear to use them. It is possible to provide extensive domain knowledge to NPCs by annotating not only objects but also the environment itself, literally using "Smart Terrain", and passing information to the NPCs about the virtual world in which they exist.

A beneficial side effect of this is that the complexity of the entities is neutral to the extent of the domain knowledge that is available for the NPCs' use, i.e.

the virtual entities themselves can not only be kept relatively simple, but they do not need to be changed at all to be able to make use of additional knowledge. If all annotated objects use the same interface to provide knowledge to NPCs then there is no limit to the scalability of the system, i.e. the abilities of NPCs can practically be extended indefinitely despite a very low impact on the system's overall performance.

9.2.1 Affordance and Annotations

Affordance theory [Cornwell et al. 2003] has its roots in psychology and the study of (visual) perception (see also Chapter 2, Section 2.3.4.4). Affordance itself is an abstract concept, the implementation of which is greatly simplified by annotations that work like labels containing instructions which provide an explicit interpretation of affordances. The relationship between affordance and annotation becomes clear when one examines the following example of a button that needs to be pressed to activate some sort of device: affordance in this case is the shape of the pressable button that "affords" to be pushed, whereas an appropriate annotation in this scenario would be a label on the button reading "press here", explicitly inviting a user to push the button.

9.2.2 Implementing Smart Environments

Annotations have been employed in several different types of applications in order to achieve different effects. Annotations have proven popular for the animation of virtual actors in computer animation, facilitating animation selection [Lee et al. 2006], i.e. the choice of appropriate animation sequences that fit the environment. Other uses of annotations include the storage of tactical information in the environment for war games and military simulations [Darken 2007], which is implemented as sensory annotations to direct the virtual entities' perception of their environment.

Probably the most common form of annotations found in computer games affects behaviour selection, often in combination with animation selection [Orkin 2006], i.e. the NPC's behaviour and its visual representation (animation) are influenced by the annotated objects that it uses. Here the annotated objects

actually have embedded instructions that are executed by the NPCs that attempt to use these objects. If this type of annotation is used, then annotations of the environment itself are implemented through invisible objects that the NPCs can interact with.

The NPCs that inhabit these annotated worlds can be built utilising a rule-based system often based on simple FSMs in combination with a knowledge interface based on a trigger system that allows the NPCs to "use" knowledge (instructions) for handling the annotated objects. The interaction protocol employed to facilitate the communication between an NPC and a "smart" object needs to enable the object to "advertise" its features to the NPCs and then allow the NPCs to request from the object relevant instructions (annotations) on its usage [Macedonia 2000]. This communication between annotated object and NPC can be achieved using techniques related to messaging [Harmon 2004].

In the extremely popular computer game "The Sims" [Kornrumpf 2005] a very similar method to the one described above is used to enable NPCs to interact with objects in the game world. Object annotations are implemented as scripts in the proprietary SimAntics programming language [Macedonia 2000]. In addition to this, Forbus and Wright [2001] state that in "The Sims" all game entities, objects as well as NPCs, are implemented as scripts that are executed in their own threads within a multitasking virtual machine. They explain that once an NPC's decision making tasks it to "use" an object which advertises a feature that will satisfy the NPC's needs, the NPC will execute the appropriate function provided by the object within its own thread. This is a similar mechanism for implementing entity annotations to the one that is available in the SEAL scripting system.

9.3 The Syntax of SEAL

The SEAL subset of AvDL is restricted to the syntactic features of AvDL that are considered essential and useful for the creation of virtual entities existing in annotated environments, i.e. the SEAL specifications only incorporate a fraction of the data types and data structures found in AvDL.

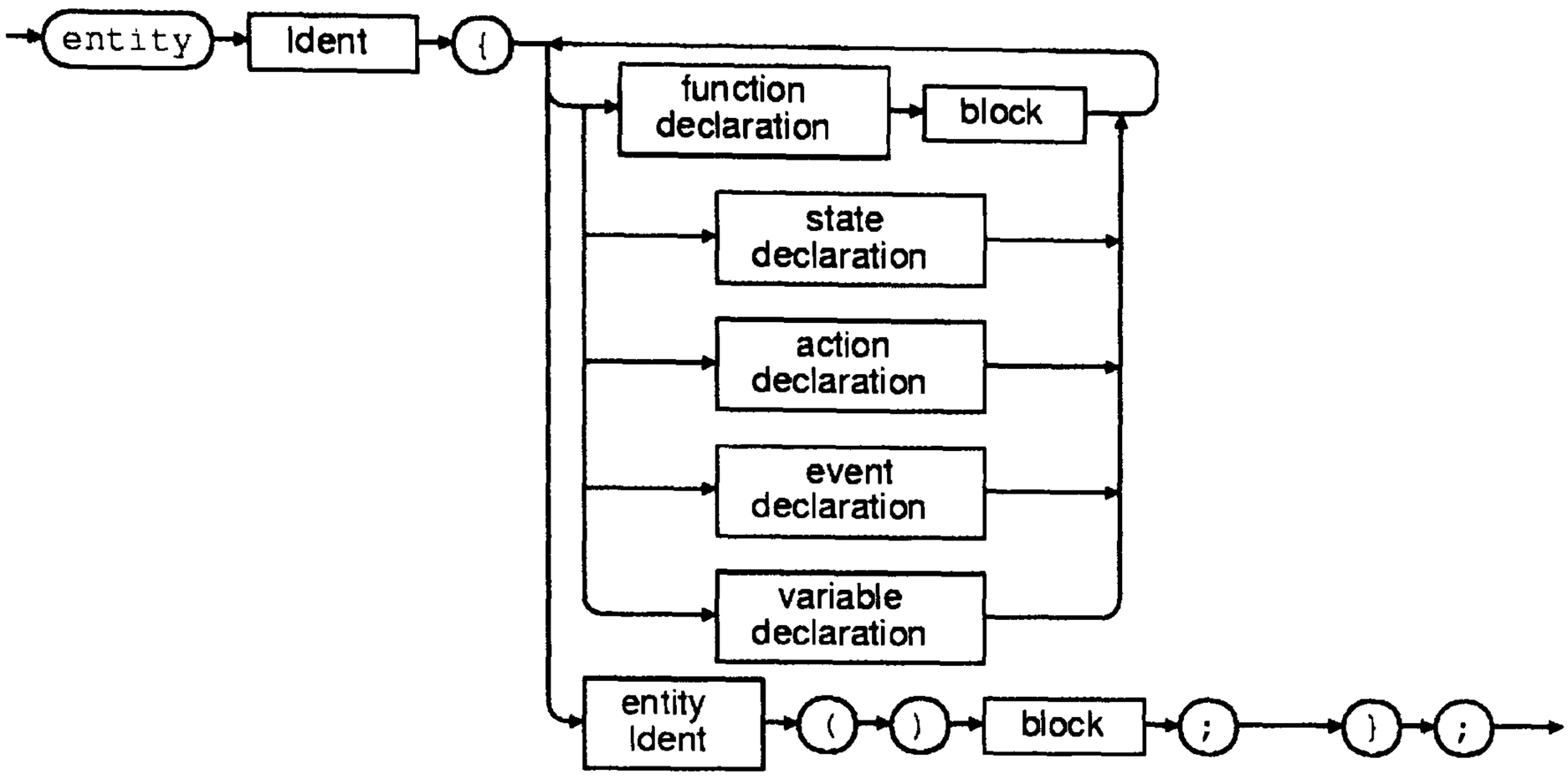


Figure 9.1: Syntax for declaring an 'entity' object.

As is the case with AvDL programs, a SEAL program is meant to encode a complete virtual entity and as such needs to declare itself as an 'entity' (see Figure 9.1). The handling of functions in SEAL is almost identical to the handling of functions in AvDL with the exception that the language specification for SEAL does not include forward declaration, i.e. there are no function prototypes for the forward declaration of functions in SEAL. The only primitive data type in SEAL is the scalar type which encodes any (binary) logical or numerical value. SEAL does not incorporate a separate Boolean data type or support type aliases or the traditional aggregate data types found in AvDL, i.e. arrays, structures or classes. SEAL therefore does not support object orientation, making the structure of SEAL programs more closely resemble the programming language C [Kerninghan and Ritchie 1988] than C++ [Stroustrup 1997]. The absence of a Boolean data type means that any values in SEAL that would be Boolean values in AvDL are scalar values set to either '1' for true or '0' for false values.

The most complex of AvDL's types remaining in SEAL is the 'state' structure, limited to the creation of finite state machines (see Figure 9.2), i.e. SEAL does not

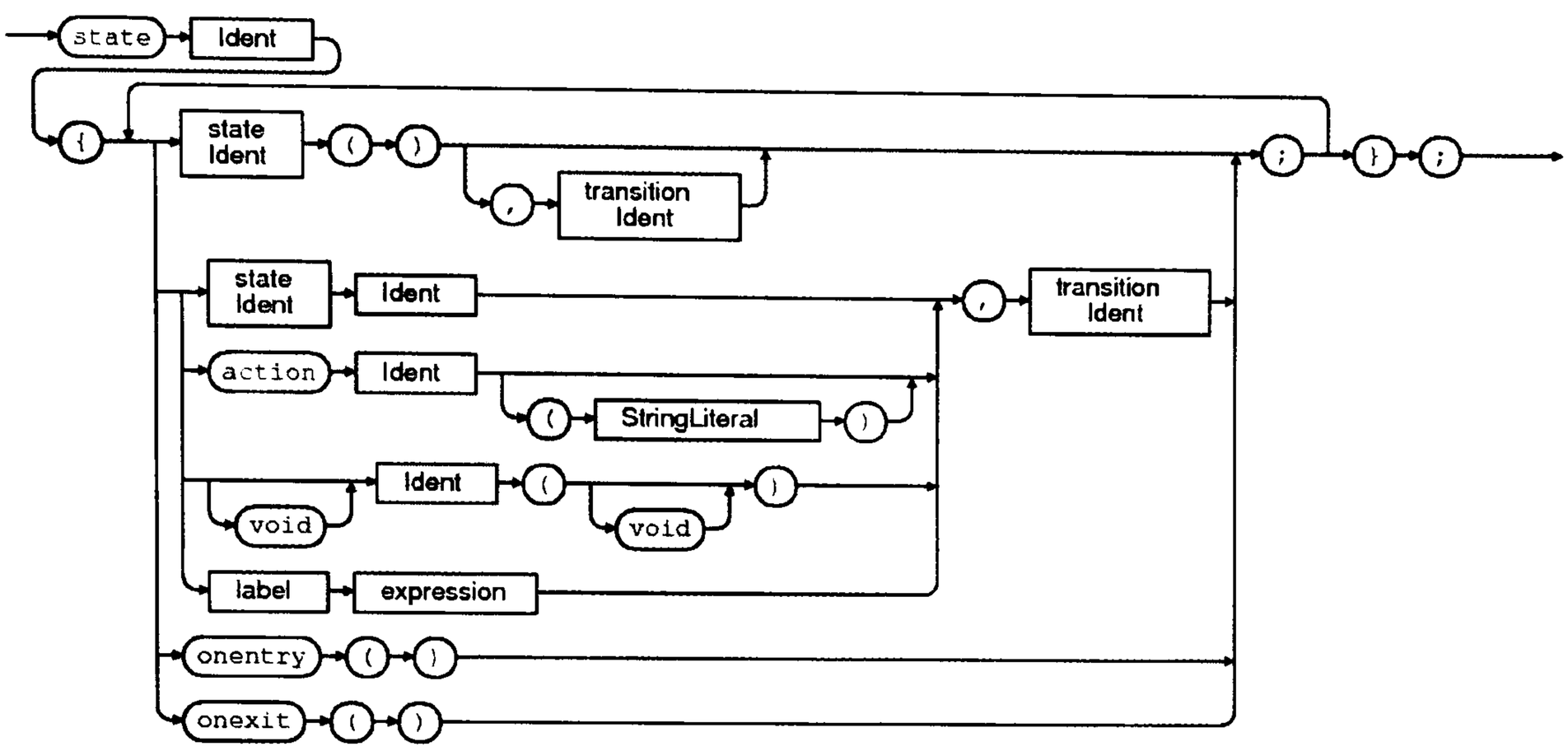


Figure 9.2: Syntax for FSM declaration.

support fuzzy state machines and therefore does not recognise the type qualifiers 'finite' or 'fuzzy', as all state machines are assumed to be FSMs. For use with 'state' data structures the operators 'getstate' for retrieving the currently set state and 'setstate' for setting the current state are also present in the specification for SEAL (see Figure 9.3), the latter operator reduced to the syntax required by FSMs, i.e. without allowing the specification of a weight value for the state.

SEAL uses AvDL's 'action' type (see Figure 9.4) to enable programs to directly call functions that are defined within the host application. The working of this type in SEAL is unchanged to that in AvDL, i.e. variables of the action type provide function bindings that map SEAL actions to functions in the host application, enabling NPC programs to directly call functions that are defined within the run-time environment's host application. As with AvDL actions, by default, mapped functions are bound to the action whose identifier corresponds with the name of the function used in its registration with the virtual machine, using the run-time environment's API (see Chapter 10, Section 10.4).

Finally, the SEAL specification also includes AvDL's 'event' type, which is used to define event handlers that can be triggered by events that occur in the

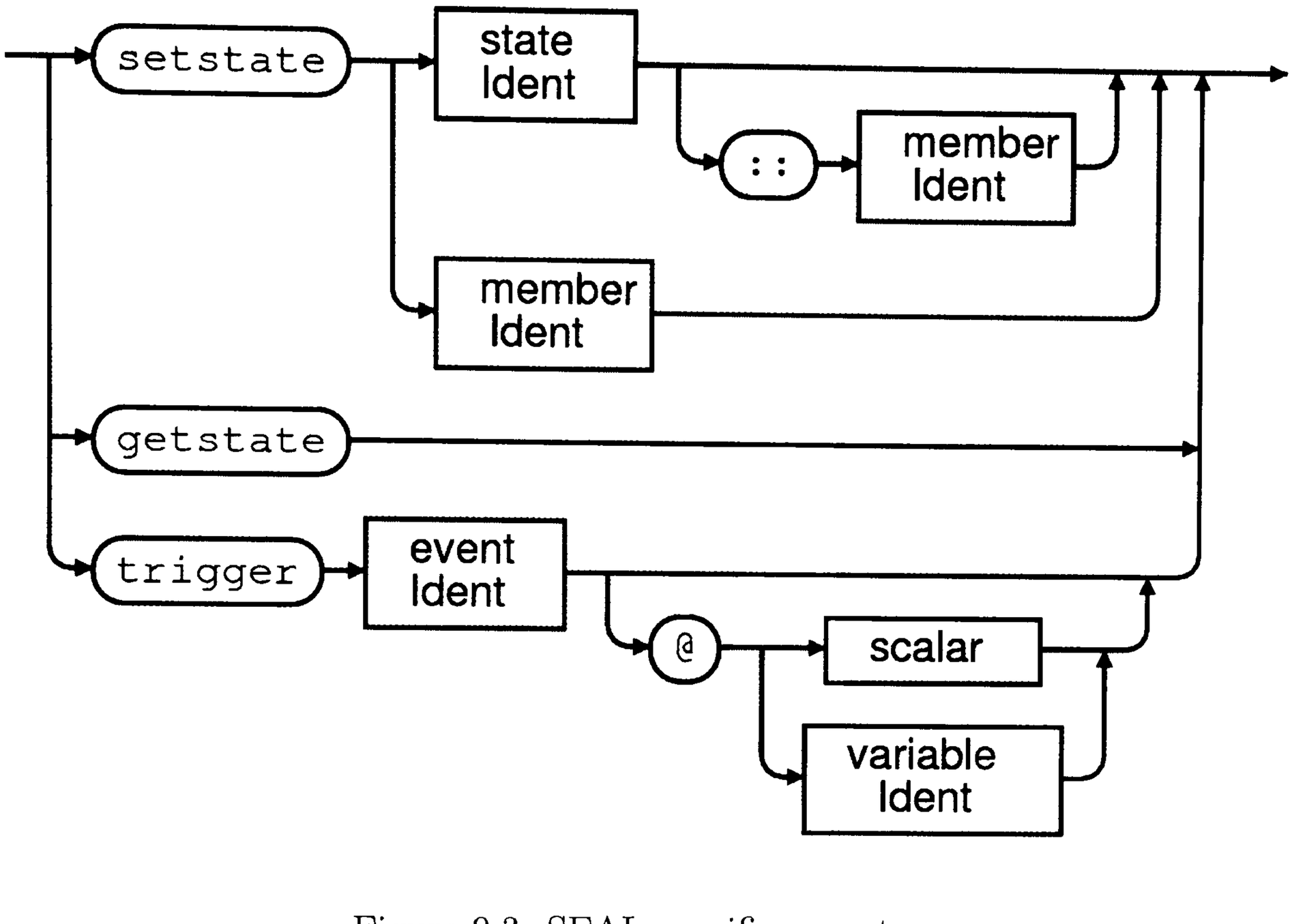


Figure 9.3: SEAL specific operators.

host application. These events can be registered with the run-time environment through the run-time API, allowing NPC programs to react to named events using the identifier provided at the registration of the event. SEAL also allows the use of AvDL's 'triggered' type qualifier for binding scalar variables to events, and also inherits AvDL's 'trigger' operator (see Figure 9.3) for spawning events from within NPC programs.

9.3.1 Entity Annotation with SEAL

All of the entities in the game world, i.e. NPCs as well as inanimate objects that can be interacted with, are defined as scripts. These are SEAL programs that are executed by a virtual machine (the system's run-time environment) which interfaces with the game engine that hosts the virtual world. The use of the

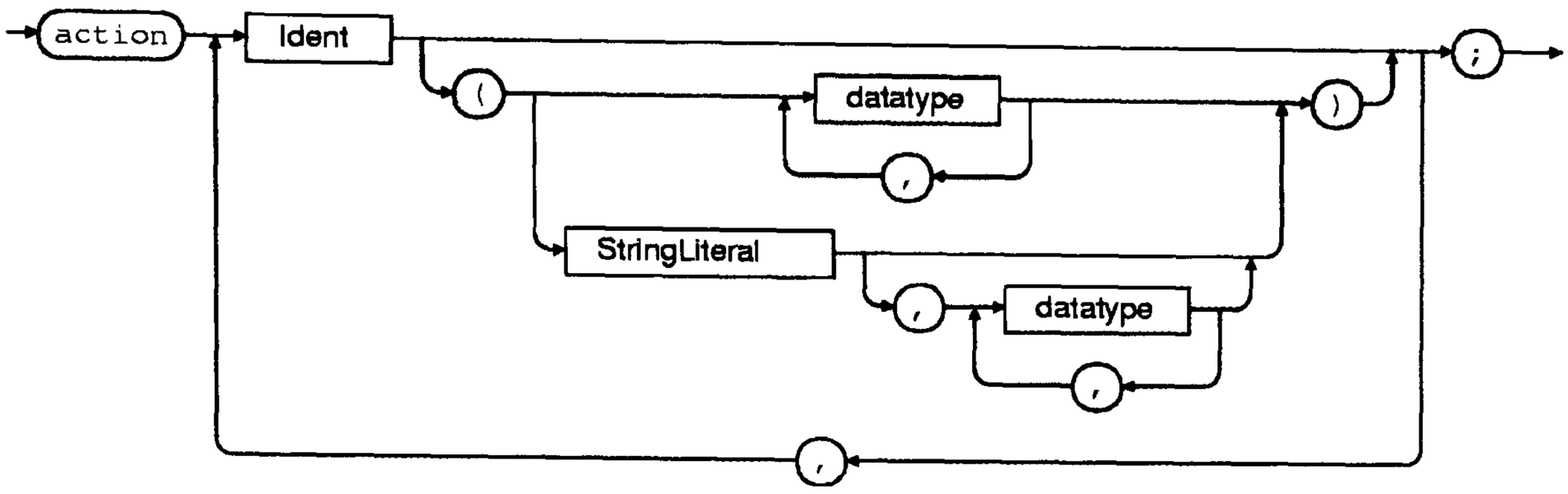


Figure 9.4: Syntax for 'action' declaration.

SEAL system provides annotations of objects with an identical representation to that of knowledge encoded within an NPC's rules, i.e. within the same type of program and using the same scripting language. In SEAL the environment itself can be annotated using intangible objects, i.e. objects that are invisible and cannot be directly interacted with but which are located within the virtual world and thus accessible to NPCs.

The system's interaction protocol needs to allow objects to advertise their capabilities, followed by NPCs then initiating contact with objects that they intend to use and the objects finally to provide access to the desired features by exposing the relevant procedures to the NPCs for execution.

SEAL's interaction protocol for annotated entities is implemented through a combination of system events in the run-time environment and a set of data type qualifiers that are part of the language specification, as well as several standard functions of the language (see Table 9.1). An annotated object is required to advertise all of the functions that it provides for the use by other entities that share its virtual environment. In SEAL this is achieved through the use of the 'global' type qualifier at the definition of functions, which then marks these functions as "exported" for use by other entities. By default any SEAL entity that includes such exported functions will advertise the availability of these functions to all entities within the virtual world that it can interact with. The use of the SEAL standard function 'setSilent' within an entity causes advertising of the entity's

Return Type	Function Name	Parameters	Description
scalar	**getEntity**	*scalar*	This function takes the unique ID of an exported function as its parameter and returns the ID of the entity that exported the function.
scalar	**getGlobal**	*constant string*	This function takes the name of an exported function as its parameter and returns the ID of a matching exported function if it exists or 'NULL' if it cannot find a match. If called from an event handler, only the entity that caused the event to be spawned will be searched for a matching exported function.
void	**setBroadcast**	–	This function asks the run-time environment to advertise an entity's exported functions.
void	**setSilent**	–	This function asks the run-time environment to stop advertising an entity's exported functions.

Table 9.1: SEAL standard functions for use with annotated entities.

exported functions within the run-time environment to be suspended. This suspension can be revoked through the use of the 'setBroadcast' standard function which requests the run-time system to resume the advertisement of the entity's exported functions.

NPCs are notified about objects that export functions through a system event that the host application must define in the run-time environment through API calls. The SEAL system is kept as generic as possible and the run-time environment does not provide a pre-defined event identifier for this purpose, as not every identifier may be appropriate for every host application, consequently an

acceptable identifier must be provided to the run-time environment. An NPC that is intended to use annotated objects needs to define an event handler for the event that alerts it to the presence of a suitable object. The event is triggered as soon as an annotated entity broadcasts its availability to the NPC. It makes little sense to trigger this event for all annotated entities at all times, so the host application needs to determine if an object is eligible for consideration by an NPC, i.e. to decide if an NPC is able to "read" an object's advertisements. To enable this, an appropriate condition, such as the Euclidean distance between the NPC and the annotated entity, associated with the event needs to be declared to the SEAL virtual machine by the host application, which is achieved through API calls to the run-time environment (see Chapter 10, Section 10.4). The system allows separate events with different conditions to be declared, providing a versatile environment for complex interaction between annotated entities.

Once an NPC has been notified of the availability of annotated objects, it needs to retrieve references to the object's exported functions that it requires to use. The use of a regular scalar variable as a function's identifier in a function call is always assumed to be a request to execute an exported function. For this, the SEAL standard function 'getGlobal', which takes a string naming the exported function as its only parameter, returns a scalar value that identifies the exported function from the entity that triggered the notification event. If no corresponding function is found in this annotated entity, the 'getGlobal' function returns the empty value 'NULL' instead.

A method that allows some limited communication between entities is the targeted use of the 'trigger' operator for spawning events in other entities. To directly address the annotated entity that contains an exported function, its identity within the run-time environment can be queried using the 'getEntity' standard function, the result of which can then be used to message the entity.

9.4 Using SEAL to Create NPCs

To demonstrate the usage of SEAL we can look at a typical scenario found in many computer games that includes the use of an object in the virtual world by an NPC, from which the workings of the system should become apparent.

This scenario is a combat simulation where an NPC has been assigned a base defence role to prevent an enemy from overrunning a friendly base. The defences of this base include turret-like gun emplacements that are implemented as annotated entities and which can be manned by human players or NPCs.

The approach of an enemy could be signalled to the NPC through an event 'enemy_detected', which would have to be registered with the host application and for which the NPC would need to provide an event handler. Assuming that the runtime-system event notifying NPCs about unmanned emplacements is named 'unused' and associated with the notification condition that the NPC is positioned next to the gun-turret, the structure of a program describing a defender NPC could look as follows in SEAL:

```
entity defender
{
  scalar manGun = NULL;

  event unused { manGun = getGlobal("use"); };

state fsm
{
  patrolling() , NULL;
  defending() , patrolling();
  fsm() , NULL;
};

event enemy_detected { setstate fsm::defending; };

fsm::patrolling()
{
while(1)
{
  /* execute 'patrolling' behaviour */
  ...
```

```
    }
}

fsm::defending
{
    if(manGun != NULL) /* if gun-turret available */
    {
        manGun(); /* man gun turret */
        ...
    }
    else
    {
        /* execute default defence behaviour */
        ...
    }
}

    fsm::fsm()
    {
        setstate patrolling;
    }

    defender()
    {
        setstate fsm;
    }
};
```

The above program shows the sections of the NPC behaviour definition that are relevant to the use of annotated gun emplacement entities. If the 'enemy_detected' event is triggered, the defender NPC is set to execute the 'defending' state. If an available gun emplacement has been found beforehand, i.e. if the 'manGun' variable has been successfully mapped to the gun-turret's 'use'

function and does not hold the empty value 'NULL', the NPC will call the exported 'manGun' function to use the gun emplacement. In this example, the script defining the gun-turret exports its 'use' function for use by other entities, which includes the declaration of an action 'fire' that maps the function to fire the turret's gun and allows it to be fired from within the exported function:

```
entity turret
{
  ...

  global void use()
  {
    action fire(); /* action to fire the gun */
    ...
    fire();
    ...
  }
  ...

  turret() {}
};
```

If the above entity scripts were used, there could be conflicts between several NPC programs that could compete to simultaneously use the same gun emplacement. To prevent this it makes sense to provide the NPCs with a means to lock the gun-turret entity while it is being used, which can be accomplished by requesting the annotated object to stop advertising its functions. This communication with the gun emplacement can be implemented by triggering events within the annotated objects, which requires the events 'lock' and 'unlock' to be registered with the host application. The refined NPC program could then be written as follows:

```
entity defender
{
  scalar manGun;
  scalar turret;
  scalar gunAvailable = 0;

event unused
{
  manGun = getGlobal("use");
  if(manGun!=NULL)
  {
    turret=getEntity(manGun);
    gunAvailable = 1;
    trigger lock @ turret;
  }
};

...

fsm::defending
{
  if(gunAvailable) /* if gun-turret available */
  {
    manGun(); /* man gun turret */
    trigger unlock @ turret;
    gunAvailable = 0;
    ...
  }
  else
  {
    /* execute default defence behaviour */
    ...
  }
```

```
    }

    ...
};
```

To make use of these improvements the turret entity needs to define event handlers for the 'lock' and 'unlock' events, as shown below:

```
entity turret
{
  event lock { setSilent(); }; /* suspend advertising */

  event unlock { setBroadcast(); }; /* resume advertising */

  ...
};
```

Chapter 10

Implementation of NPC Programs on the System's Run-Time Environment

The previous two chapters provided an overview of the AvDL scripting language (see Chapter 8) and its SEAL subset (see Chapter 9) and discussed the features of these two languages. This chapter presents the design of the SEAL/AvDL run-time environment, i.e. the virtual machine for executing programs that encode virtual entities, as well as the implementation of the system prototype and the interface to the virtual machine that allows it to be embedded within a host application.

The virtual machine of the system prototype, while based on the SEAL specification, makes provision for a large proportion of the features described in the AvDL specification, such as the implementation of the system's extension architecture which itself is not part of the SEAL specification.

The current prototype system includes an assembler that is capable of generating bytecode programs that utilise all of the features that have been implemented in the virtual machine, allowing the use of "hand-translated" programs in the absence of a working compiler for the system. Some details regarding the translation of features of the SEAL and AvDL into instructions for the run-time environment are presented below.

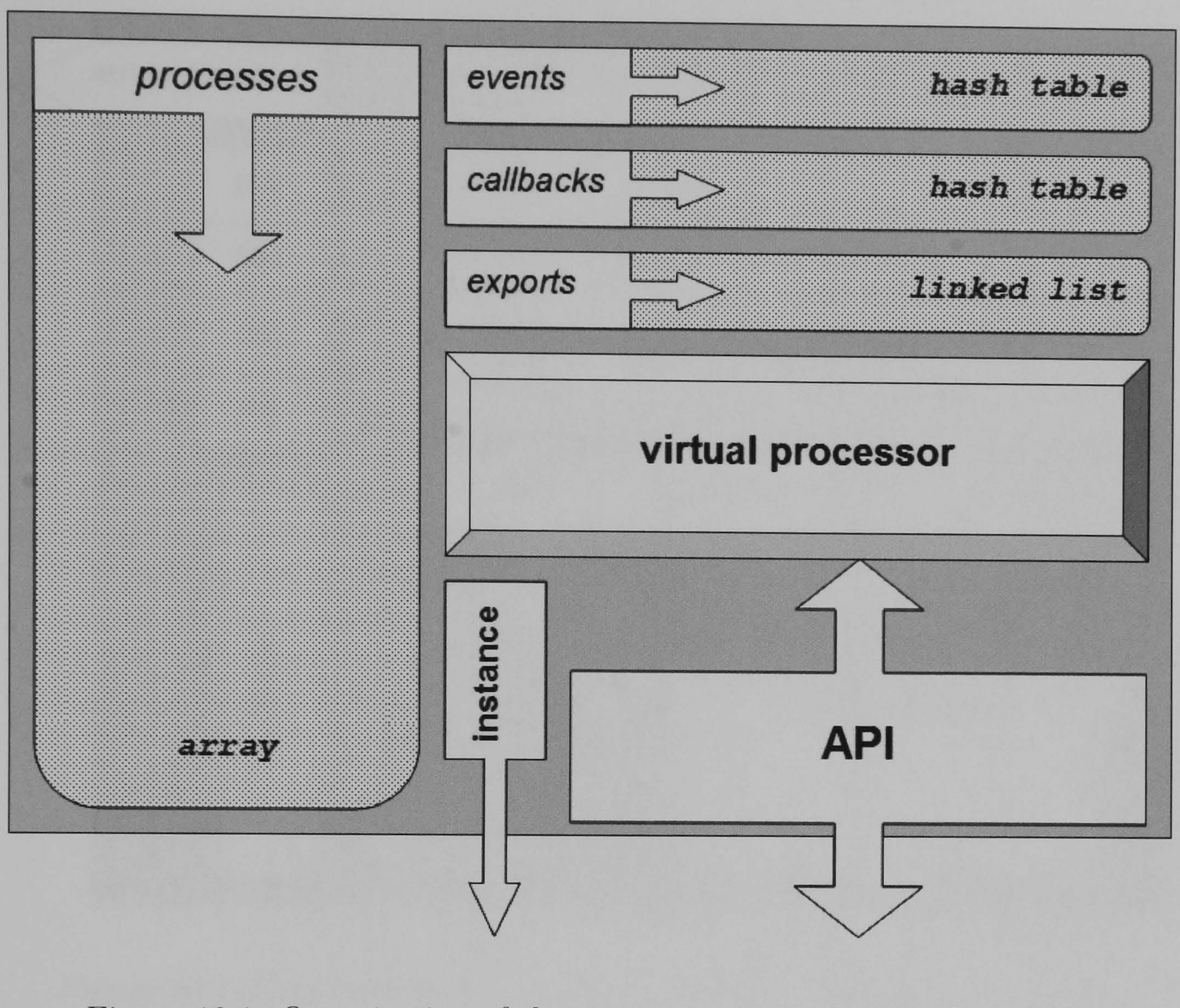


Figure 10.1: Organisation of the system prototype's virtual machine.

10.1 Virtual Machine Architecture

The architecture of the system is based on our ZBL/0 [Anderson 2004] and C-Sheep [Anderson and McLoughlin 2006] virtual machines (see Figure 10.1). At its core the system's virtual machine has a parallel stack machine, which, with the exception of the extension architecture, is written in platform independent ANSI C++. Similar to its predecessors the system's prototype allows the creation of several simultaneously running processes, but unlike the earlier virtual machines that held all data within a single object, here each process is a separate object, keeping different programs separate from each other.

The virtual machine object provides the API for integrating the system in its

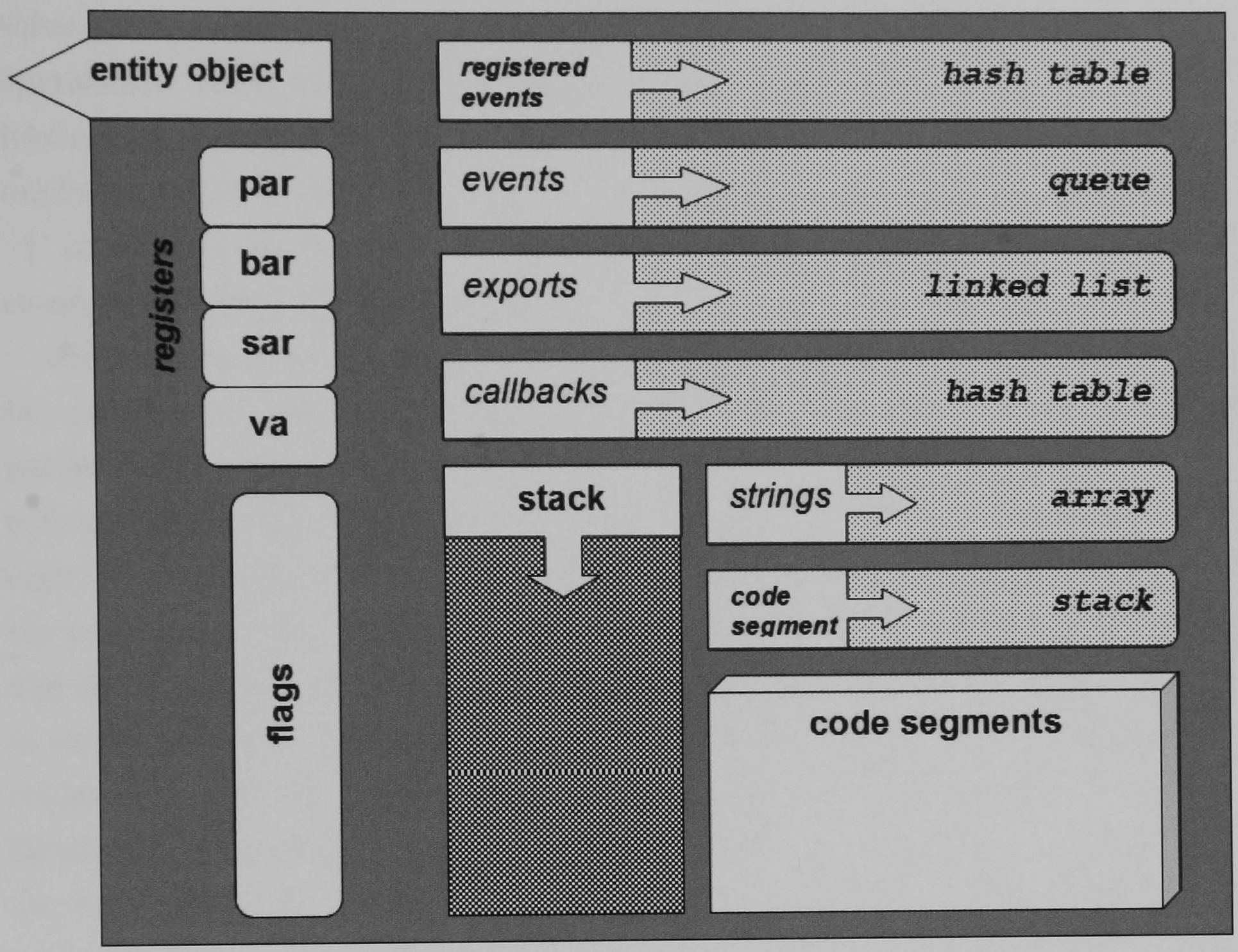


Figure 10.2: Organisation of an entity's process in the system prototype.

host application, as well as the system's virtual processor and an architecture for managing extensions to the system and communication between processes, but all data required by the processes themselves is kept safely encapsulated within the process objects.

A process object provides a separate entity, embedding its own stack, registers (program counter, program instruction register, base address register and stack register) and code segments, resulting in each entity effectively providing a self-contained micro-thread [Dawson 2001] in the virtual machine (see Figure 10.2).

Inspired by the architecture of the 80x86 processor family [Link 1995], data entries on a process's stack can be split into a high-segment and a low-segment that can be addressed separately, each of which has half of the bit-width of a data entry. This allows each data entry to hold not only single values but also

value pairs, a side effect of which is that NPC programs require fewer stack access operations. These value pairs are mainly used as parameters and return values for intrinsic functions of the virtual machine. A further use of this mechanism is the implementation of the value 'NULL' which utilises a value pair, placing the value '-1' in the high-segment as well as the low-segment of variables to distinguish the empty value 'NULL' from the numeric value '0'.

Each process can hold several code segments, allowing functions that can be exported for use by other entities to reside within segregated areas of their parent entity's process, i.e. in a separate code segment. Furthermore, a stack of references to code segments, the top of which is used as the currently active code segment, is maintained by each process. These references are not restricted to the process in which they are stored, allowing code segments to be shared among and to be accessed simultaneously by several NPC programs. As they are stored in separate code segments, a simple overriding mechanism allows functions to be replaced by different functions while the system is running, provided that the function is not being called at the time. The system prototype contains hooks for the future integration of an AOT (ahead-of-time) compiler which will allow NPC programs to be compiled just before they are loaded into the virtual machine. This will eventually also allow functions within these programs to be replaced interactively during run-time by utilising parts of this compiler as a type of OTF (on-the-fly) compiler.

The system's virtual machine is a self-contained module and accessible by a host application solely through API calls (see Section 10.4). From the outside, i.e. to the host application, several processes appear to run simultaneously on the virtual machine, whereas actually NPC programs are executed sequentially. The mechanism that allows the virtual machine to execute several NPC programs that are seemingly running in parallel is pre-emptive multi-tasking combined with round-robin scheduling. The virtual machine's execution cycle itself proceeds in two stages, the first of which is the event handling cycle. During this stage all events from previous execution cycles that have not been handled, as well as those that have been triggered during the preceding execution cycle of the virtual machine, are acted upon and the event handlers within each of the loaded processes are executed (see Section 10.2.2). To prevent synchronisation conflicts,

if any events for a process remain unhandled during the virtual machine's execution cycle, the corresponding process will be blocked until all events have been processed in a subsequent execution cycle of the virtual machine.

Once all events have been processed, then the second stage of the virtual machine's execution cycle (i.e. its regular run cycle) commences. This executes the instructions found in the currently active code segment which is referenced at the top of the process's code segment stack.

Following the example of its predecessor systems, the design of the virtual machine is inherently fault tolerant. Run-time errors, resulting from illegal memory access operations within NPC programs or program instabilities caused by faulty interactions between NPC programs, will only result in the termination of the offending processes without affecting other programs running on the virtual machine or the operation (i.e. functioning) of the virtual machine itself. Thus , like its predecessor system, the virtual machine should degrade gracefully.

10.1.1 Virtual Machine Instruction Set

The instruction set of the system's virtual machine is an extended version of the C-Sheep virtual machine's instruction set (see Chapter 7, Section 7.3.2.1). Similar to the way in which the C-Sheep system works, all numerical data values that are handled by the prototype system are of the same data type, leaving distinctions between types and the maintenance of type-safety aside as an issue to be dealt with by a compiler.

The instructions of the prototype system fall into several categories (see also Appendix F). The first of these is process control, including instructions that direct program flow and memory management, which also forms part of the tasks of the second category that includes instructions for data handling, i.e. access to variables and memory addresses. The third category is made up of instructions for the use of functions, including intrinsic system functions, extension functions (see Section 10.1.2) and user-defined functions. The penultimate two categories are comparisons and operators, facilitating the processing and manipulation of data on the process's stack. In addition to these categories, op-codes for memory manipulation instructions to access a heap for dynamic allocation of data storage

Function Name	Description
executeCallback	System function that executes a callback function.
getExported	System function that retrieves a reference to an exported function whose location is unknown.
getFuncAddr	System function that retrieves a reference to an exported function from a known entity process.
retrievePID	System function that retrieves the process ID of the current entity process.
setBroadcast	System function that asks the virtual machine to advertise the process's exported functions.
setSilent	System function that asks the virtual machine to stop advertising the process's exported functions.
spawnEvent	System function that allows an entity process to trigger an event in the virtual machine.
stateTransition	System function that sets a process flag to trigger a state transition at the execution of the next instruction.

Table 10.1: Intrinsic system functions of the prototype's virtual machine.

have been reserved within the system, but have not been implemented for the current prototype (see Chapter 12, Section 12.3), as they are not required for programs that adhere to the SEAL specification.

Other than virtual machine instructions, the system prototype also provides several low-level system functions that are directives to the virtual machine which are implemented as intrinsic functions of the system (see Table 10.1).

10.1.2 Extension Architecture

The extension architecture of the AvDL system is directly based on the extension architecture of the ZBL/0 virtual machine (version 1.2) that allows the extension of the system through plug-ins [Anderson 2004] (see also Chapter 7, Section 7.2.5). These plug-ins (extension libraries) themselves are implemented as shared objects (libraries) that can be dynamically loaded during program run-time.

Each plug-in contains the definition of a class for the extension which it inherits from an extension class that is part of the system's API. The derived class that makes up the plug-in must implement a set of interface methods that allow the virtual machine or other system components, such as a compiler, access to the extension library. The first of these methods returns the number of useable functions contained in the extension library. Analogous to this method, there are also methods for retrieving the number of constant values and operators, provided by the plug-in, for use in NPC programs. Other methods can be used to obtain the signatures (return type, identifier and formal parameters) of functions provided by the plug-in, as well as the identifiers and values of constants. New operators that are provided by plug-ins are treated very much like a special case of an extension function.

10.1.2.1 Extending the Language at Compile-Time

The process that enables a compiler to process programs that utilise extension libraries is straightforward. When compiling NPC programs, the compiler first needs to load in any extension libraries that are requested by the program that is being compiled, which the program can achieve by using the language's 'import' statement. Each extension library is then assigned an ID by the compiler which identifies the plug-in within the NPC program. This value is stored within the compiled bytecode of the NPC program with a reference to the file-name of the plug-in. The next step that a compiler then takes is to determine the number of functions provided by the plug-in, after which it can query and then add each of these functions to its identifier table, allowing the compiler to verify the syntax of function calls to extension functions. This step is then repeated for constants and operators that are contained in the extension library.

During code generation, function calls to functions in the plug-in are stored in the NPC program bytecode with a reference to the extension library ID value and the index value that identifies the extension function itself within the plug-in.

10.1.2.2 Extending the System at Run-Time

Plug-ins are managed centrally within the system's virtual machine. When a compiled NPC program is loaded into the virtual machine, the virtual machine also dynamically loads in all previously unloaded extension libraries that are required by the NPC process (plug-ins that are referenced in the program). If the virtual machine fails to find and load all of the plug-ins that a program requests, then the NPC program itself will not load and the NPC process will not be created. If the virtual machine succeeds in loading the plug-ins or finds them among already loaded plug-ins, then the plug-in ID values that were stored within the NPC program's bytecode are mapped to the actual plug-in ID values that are used within the virtual machine.

During the NPC program execution all extension library calls are then redirected to the plug-in whose mapped value corresponds to the ID from the compiled bytecode program. This is achieved by executing the 'call extension function' instruction, which grants the extension library access to the process's stack, referencing the plug-in ID as well as the index value that identifies the extension function inside the plug-in.

10.2 Implementation of the System Prototype's Features

The system prototype and its virtual machine described here implement the features described in the specification of the SEAL BDL (see Appendix E). The implementation of data types and data structures that are not commonly represented in implementation programming languages, i.e. features that are specific to SEAL and AvDL, is detailed below.

10.2.1 Implementation of Actions

Actions, i.e. functions that exist within the host application that can be invoked from within NPC programs, are effectively callback functions which in the current version of the system prototype are implemented in the system's API through the

use of an abstract base class (callback) from which classes can be derived that contain the actual functions that the actions are supposed to be mapped to.

The definition of callback functions that are placed within these derived classes must be accompanied with the definition of the derived object's method 'execute', which is used to redirect action callbacks to the relevant functions in the host application. While this mechanism imposes some restrictions on the creation of callback functions, it is an improvement on the realisation of interaction between scripts running in the virtual machine with the system's host application that was employed in the ZBL/0 system [Zerbst et al. 2003]. In the ZBL/0 system, extensibility was limited by the necessity to use intrinsic functions. The type and number of intrinsic functions that could be registered with the virtual machine had been established by the system's API (see Chapter 7, Section 7.3.6), leaving no room for the introduction of additional functions.

The system prototype provides API functions for the registration of these callback functions with the virtual machine, allowing the association of the names of actions with the corresponding object that was derived from the 'callback' class. For this, the virtual machine provides two levels of access to actions that are maintained in separate lists of callback functions. The lower level of access to callbacks is managed by the virtual entities' processes themselves, allowing actions to be registered directly with a specific process. The higher level resides within the virtual machine itself, allowing callbacks to be registered globally with the run-time environment and providing a fall-back to "default" actions if no appropriate callback has been registered with a process.

The invocation of actions is translated to a call to the intrinsic system function 'executeCallback' that first attempts to find a corresponding callback within the list of callbacks, which are registered with the current process. If no appropriate callback can be selected, a callback that fits the action will be sought from the virtual machine itself, resulting in a run-time error if the virtual machine fails to find a callback associated with the action. If a callback is correctly identified, an intermediate call-stack data record is created and filled with the action's parameters (retrieved from the process's stack) using the signature (i.e. return type and formal parameters) that the callback function was registered with. The callback function is then executed and if the action is expected to return data to its caller,

then the call-stack record is augmented with the action's return value which is subsequently placed onto the process's stack.

10.2.2 Implementation of Events

Events are registered globally within the virtual machine through the run-time environment's API that provides a unique ID for every event that is added to the virtual machine. Each NPC process also maintains a list of events that it can handle or spawn, which is automatically generated when an NPC program is loaded. This event list maps the process's internal representation of events to the unique IDs of events that are registered with the virtual machine. Within the NPC process, a variable data entry of global scope is created at the bottom of the process's stack for every event handler or event that the process can trigger, allowing the use of the event name as a variable identifier in NPC programs. This data entry holds a value pair. In its high-segment it holds the unique event ID within the virtual machine and in its low-segment the process ID that caused the triggering of the event or alternatively the value '-1', if the event was spawned by the system.

10.2.2.1 Event Handlers

In a similar manner to exported functions (see Section 10.1), an NPC program's event handlers are stored in code segments that are separate from the rest of the program. This is reflected in the NPC program bytecode, where event handlers are enclosed with the 'mark handler start' and 'mark handler end' virtual machine instructions. The former of these instructions is never interpreted by the virtual machine while a loaded NPC process is running, as it is only used while the program is loaded to instruct the virtual machine to provide a separate code segment for the event handler.

The data held within each NPC process includes a queue data structure for all events that the process can handle which have occurred and that have not yet been handled. During the event handling cycle of the virtual machine's execution cycle, each data entry of this queue is retrieved and processed (i.e. the event handler that each event is associated with is invoked, handling the event). For this the

virtual machine pushes a reference to the code segment, which corresponds to the event handler, onto the top of the process's code segment stack. This operation makes the event handler the currently active code segment for execution. An NPC program's event handlers are executed asynchronously, i.e. separate from the rest of the program, making them similar to coroutines. As such, the first instruction in every event handler creates a block activation record on the process's stack before any other instructions are processed. The 'mark handler end' virtual machine instruction, which is the final instruction of every event handler, cleans up the stack and if there is an unhandled event left in the event queue it replaces the code segment of the current event handler on the code segment stack with the event handler of the next unhandled event. If no more events remain in the event queue, the code segment stack is reset to the last active code segment used by the NPC program, allowing the process to resume its execution during the regular run cycle of the virtual machine's execution cycle.

10.2.2.2 Event 'trigger' Operator

The 'trigger' operator for setting off events is implemented as a sequence of data handling instructions that load the event data entry and optionally the ID of the event's target process onto the current process's stack, followed by the invocation of the 'spawnEvent' intrinsic system function.

10.2.2.3 'triggered' Variables

Like the event data entries themselves, and similarly to variables that have been declared using the 'volatile' type qualifier, variables that are declared as 'triggered' by an event have their memory allocated at the start of the program's execution. Consequently they are also stored at the bottom of the stack, so that the run-time environment is aware of their location on the stack, allowing it to update them when an associated event occurs. Reading data from this type of variable will alter the variable's content to the value '0' or 'false', as once the variable has been read the event will be considered as having been handled, requiring the variable to be reset.

10.2.3 Implementation of FSMs

The most popular data structure used for the creation of intelligent NPCs is the finite state machine (see Chapter 8, Section 8.2.1.3.1), which is part of the SEAL specification, as well as the AvDL specification. NPC programs that include FSMs require the creation of two data entries at the bottom of the process stack at program start. The first of these data entries, which is given an initial value of 'NULL', references the current state that the program's state machine finds itself in, whereas the second data entry holds a reference to the next state that the program's state machine is expected to transition into.

The data entry for the current state is used for querying the status of FSM structures or structure members, i.e. for determining if a state is currently active or inactive. This type of query operation is not implemented using any special operators or system functions, but instead it is translated to a sequence of regular virtual machine instructions that simply compares the current state to the state that is being queried.

10.2.3.1 FSM Specific Operators

A call to the 'setstate' operator is mapped to an implicit function call to an unnamed function, which holds a sequence of virtual machine instructions that first change the next state data entry to the new transition target and then call the 'stateTransition' intrinsic system function that will trigger the state transition at the start of the execution of the next instruction during the regular run cycle of the process. In the implementation of our prototype virtual machine, the data returned by this unnamed function (the 'setstate' operator's return value) is simply a constant value referencing the state that is being set.

Querying the currently active state, by using the 'getstate' operator, is translated to a simple data handling instruction that loads the content of the data entry which references the current state of the FSM.

10.2.3.2 Program Flow in FSM Structures

The result of the translation of FSMs to the system's virtual machine instructions (see Appendix F) bears some similarity to the structure of a sub-program or

subroutine that has been translated into instructions for the virtual machine. As program flow is diverted to the state machine until it terminates, all state structures within an entity program are handled as if they were part of a single function, i.e. program flow between states is not nested (as would be the case with functions), but sequential (i.e. states are executed one after the other and a new state is only entered after the previous state has finished execution). Furthermore, whereas in the SEAL or AvDL source code a state structure's special methods (entry, exit and state body methods) have the appearance of functions (in terms of virtual machine instructions), they actually translate into regular branching of program flow using jump instructions (inspired by code expansion results of the macro-based state machine language proposed by Rabin [2002b]).

Every SEAL or AvDL program that utilises FSMs includes a sequence of instructions to which program flow is diverted after a state transition has been triggered. In this case, if the current state data entry holds the value 'NULL' (i.e. if there is no active state), then before the program flow diversion occurs a block activation record (which is similar to that found in functions) is created on the process's stack to ensure that the process can be restored to its original state when the execution of the FSM finishes. After the block activation record has been created, the program will jump to the first instruction of the newly set state's entry method. If there is an active state, however (i.e. if the current state data entry holds a reference to a state that has been set), the program will jump to the first instruction of the current state's exit method to start the transition to the newly set state.

If the transition target of a state holds the value 'NULL', a sequence of instructions that are an implicit part of every state's exit method will remove the state machine's block activation record. As a result, the virtual entity's process program flow will return to the statement that was being executed just before the state machine was first initialised (i.e. when the 'setstate' operator was first used to activate the FSM). If a different target state has been set, then program flow will branch to the next state's entry method instead.

Once an entry method has been entered, first the current state and the next state data entries are set to the new current state (i.e. the previous state's transition target) and the new current state's pre-defined transition target respectively,

before any other instructions of an explicitly defined entry method are processed. The last instruction of every entry method will always be a jump to the first instruction of the state's main method, i.e. the state's body.

The final instruction of every state's body calls the 'stateTransition' intrinsic system function, triggering the FSM's transition to the state referenced in the next state data entry at the start of the execution of the next instruction.

Within a state machine structure, the structure's members are treated similarly to state structures and are mapped to sequences of instructions that reflect those encoding the state structures themselves. Member functions of state structures are translated to a regular function call embedded within the implicit entry and exit methods. That same mechanism, of using implicit entry and exit methods, is used for a state's labelled expressions and 'action' members of state structures.

10.2.4 Implementation of Entity Annotation

Like event handlers that need to be executed separate from the rest of the entity programs that they are defined in (see Section 10.2.2), exported functions are stored in separate code segments within an entity's process. This separation allows other entity programs to access them (see Chapter 9, Section 9.3.1). Within entity program bytecode, functions that are marked as exported with the 'global' type qualifier are enclosed with the 'mark exported start' and 'mark exported end' virtual machine instructions. The 'mark exported start' instruction is only used when a program is loaded to direct the virtual machine to provide a separate code segment for the function and therefore it is an instruction that is never interpreted by the virtual machine while a loaded NPC process is running. Whereas most of the instructions used in exported functions are identical to the instructions used elsewhere in entity programs, exported functions use the 'return from exported' instruction when returning to their caller instead of the regular return instruction. The 'mark exported end' instruction is the last instruction in every exported function and will return program flow to its caller if the function does not include a separate return instruction.

Most of the standard functions that are used for entity annotation are mapped to intrinsic functions of our prototype system as they are directly interacting with the run-time environment, while a few standard functions are implemented as sequences of regular virtual machine instructions. Within the NPC program that uses an entity's exported functions, the SEAL/AvDL standard function 'getGlobal' is used to obtain a reference to an exported function.

This process is implemented by first loading the exported function's name (as a constant string) and the ID of the process that exports the function onto the stack and then by calling the 'getFuncAddr' intrinsic system function that returns either a value pair of process ID and code segment index, which references the exported function, or the value NULL if no such function exists. This resulting value can then be stored in a scalar variable that can subsequently be used as an identifier for the exported function.

The 'getEntity' standard function that retrieves the ID of a process that advertises an exported function, however, is not mapped to an intrinsic function. Instead, it is translated into a simple data handling instruction that retrieves the low-segment containing the process ID from the scalar value that references the exported function.

The system also makes provision for using an exported function whose parent process is unknown – a mechanism that is used for the 'getGlobal' standard function of the AvDL or SEAL systems if the function is invoked within the main program code, rather than an event handler. In that case the 'getExported' intrinsic system function is called. This function will search all exported functions that are known to the system for the first function that matches the requested identifier and returns a reference to the function as a scalar value, optionally allowing the specification of the requested function's signature (i.e. its return type and formal parameters).

Once a reference to an exported function is available, it can be used to execute the exported function. For this to happen, if the exported function expects parameters. then first all of the function's parameters need to be loaded onto the stack, after which the variable that references the exported function must be loaded onto the stack. Subsequently the instruction to execute an exported function is invoked, which will push a reference to the code segment that holds

the exported function onto the process's code segment stack. This will make the exported function's code segment the currently active code segment for execution which will direct program flow to the instructions of the exported function. If during the execution of the instructions of the exported function no invocation of the 'return from exported' instruction is encountered, then eventually the 'mark exported end' instruction, which is the last instruction in the code segment of every exported function, will be executed. Like the 'return from exported' virtual machine instruction it will remove the exported function's code segment from the code segment stack and restore the last active code segment used by the NPC program at the top of the process's code segment stack, allowing the process to resume its regular execution.

10.3 Considerations for Extension to Full AvDL Specification

The system prototype presented here was designed to implement the features described in the SEAL specification. Consequently the prototype does not make explicit provision for object orientation, which is mainly a compiler issue (see Chapter 12, Section 12.3), as the virtual machine's instruction set should already be capable of handling programs that utilise features that are beyond the SEAL specification and that are actually part of the AvDL specification, such as object oriented AvDL programs (see Table 10.2).

The same is mostly true for the implementation of arrays, i.e. statically allocated arrays could already be implemented on the existing prototype. Dynamically allocated arrays, including associative arrays, however, would require the implementation of the memory manipulation instructions for which op-codes have already been reserved (see Section 10.1.1).

AvDL source code	instructions
	Constructor:
	`isa 3`
	`ldc 5` *# load value for data1*
	`ldc 0` *# load offset for data1*
	`lod 0 -1` *# retrieve "this" pointer*
	`add` *# add offset to address*
	`sta` *# store data to address*
	`ldc 2.5` *# load value for data2*
	`ldc 1` *# load offset for data2*
	`lod 0 -1` *# retrieve "this" pointer*
	`add` *# add offset to address*
	`sta` *# store data to address*
	`ret $1`
	Method "method":
	`isa 4`
	`ldc 0` *# load offset for data1*
	`lod 0 -1` *# retrieve "this" pointer*
	`add` *# add offset to address*
	`lfa` *# load data1*
	`ldc 0` *# load offset for data2*
	`lod 0 -1` *# retrieve "this" pointer*
	`add` *# add offset to address*
	`lfa` *# load data2*
	`add` *# add data1 and data2*
	`str 0 3` *# store in retval*
	`lod 0 3` *# load retval*
	`ret 1 $1` *# return retval*

```
class object
{
    scalar data1;
    scalar data2;
    object();
    scalar method(void);
};
```

```
object::object() // constructor
{
    data1=5;
    data2=2.5;
}
```

```
scalar object::method(void)
{
    scalar retval;
    retval = data1+data2;
    return retval;
}
```

Table 10.2: Translation example for an AvDL class.

10.3.1 Considerations for FuSM Implementation

An implementation of the tentative FuSM type described in the AvDL specification (see Chapter 8, Section 8.2.1.3.2) could take the form of a sort of globally accessible record data structure on a process's stack. This data structure could hold as its first entry a reference to the state itself (i.e. the weight value of the fuzzy state structure itself), followed by entries for the data members of the state structure.

The 'setstate' operator for defining the degree of activity of fuzzy states would be translated to simple data handling instructions, capping the state values between '0.0' and '1.0' and storing the result within the data entries that correspond to the states that are set on the process's stack.

Similar to the querying of states in FSMs, the querying of a state's value in FuSMs is not implemented using any special operators or system functions but instead it is translated to the simple retrieval of the data stored in the state record's data entry on the process's stack.

As the mapping of these data entries to their corresponding states is a translation issue that would need to be addressed by a compiler for AvDL programs, the implementation of this type of FuSM would not require the extension of the virtual machine with additional instructions or intrinsic system functions, but could be realised with the features of the current system prototype.

10.3.2 Considerations for Goal Implementation

The implementation of goal-oriented action planning using the method described by Orkin [2004a] would require the extension of the system's process data structure to include a list of goal nodes that would need to contain the goal's priority (weight) value as well as a means for storing links to other goal nodes.

During the loading of an entity program, each goal defined in the process's entity program would then be added to this list of goals, defining the search space for a planner that would be invoked by the use of AvDL's 'plan' operator. This planner itself would be implemented as a dedicated search method, provided to the virtual machine and utilising the A* algorithm as presented earlier in this

thesis (see Chapter 4, Section 4.3.3). complemented with a different cost function based on the priority values assigned to each goal.

In addition to the implementation of the planner itself, the run-time environment's API could also be provided with a means to allow the registration of an alternative search function with the virtual machine, i.e. a planner interface to allow for greater customisation by enabling the host application's developer to implement a custom planning method.

This could be achieved by using a mechanism based on the implementation of callback functions (see Section 10.2.1) for this planner interface, similar to the way in which comparisons in search and sort functions are implemented in the C programming language's standard library [Prinz and Crawford 2006].

10.4 Interfacing a Host Application with the System

The prototype's run-time system can be integrated into a host application that creates a virtual world that can be inhabited by virtual entities, i.e. typically a game engine. For this the system's virtual machine, which manages entities that have been defined by programs written in AvDL or SEAL, provides an API that allows the host application to access the run-time environment.

10.4.1 The System API

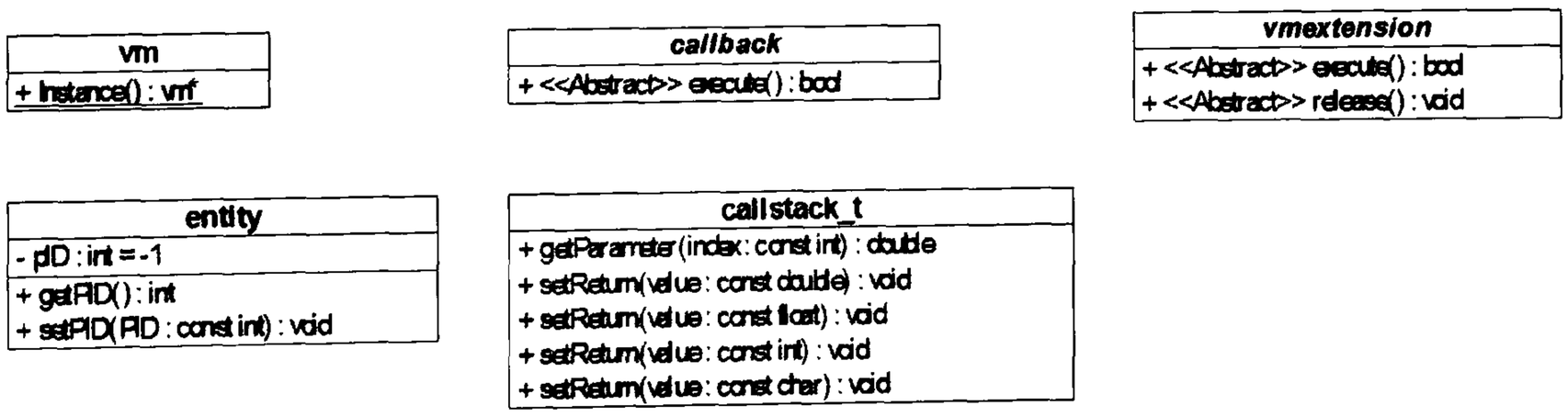


Figure 10.3: The classes of the run-time environment's API.

The API of the prototype system's run-time environment is made up from several classes (see Figure 10.3) that provide methods for loading, executing and influencing virtual entity programs in the virtual machine.

The purpose of some of these classes is to facilitate the definition of callback functions and the binding of virtual entities in the virtual world of the host application to their corresponding entity processes that are running on the virtual machine, using the multiple-inheritance functionality of the C++ programming language [Stroustrup 1997]. A virtual entity class from which the entities that populate the host application can be instantiated and that is to be associated with a SEAL or AvDL program must be derived from the API's 'entity' base class. Any class that includes functions that are supposed to be used as callback functions for the virtual machine needs to inherit from the API's abstract 'callback' class and must implement the derived class's 'execute' method. Within this method, the callback's call-stack can be accessed through methods of the API's 'callstack_t' class.

Other classes of the API provide the management architecture for extension libraries, as well as templates for the creation of extensions (see Section 10.1.2).

The API's main point of access, however, is the virtual machine object itself, as it contains the functions that are essential for the operation of the run-time environment. This object is an instance to a class that employs the singleton (design) pattern [Boer 2000], i.e. an object of which only a single instance can exist and which is accessible solely through the method 'Instance'.

The API contains different types of interface methods (see Appendix F) that allow the host application to interact with the system prototype's run-time environment. First of these are the methods that allow the initialisation of the virtual machine and the setting up of virtual entities, including the loading of entity programs, the registration of events and the registration of callback functions. This also includes the registration of special events, such as the export notification event and its associated callback function that is used to determine if a process is eligible to be notified about the availability of another entity's exported functions. This type of method also includes the main scheduling method (named 'run') that should be invoked for every update cycle of the host application, which is usually once for every rendered frame.

Another type of method are the virtual machine's housekeeping functions that provide non-essential functionality, which are limited to the identification of the system's version in the current prototype.

Then there are API methods that allow the host application to influence running entity processes, including among others the triggering of events in the virtual machine, which can be achieved by using the 'spawnEvent' method, as well as methods for altering a process's priority in the virtual machine, which can be useful if the host application attempts to implement some sort of entity level-of-detail (LOD) operations [Brockington 2002].

This last type of API method also includes the API method 'setValue', which is used to alter the content of variables that have been marked with the 'volatile' type qualifier in SEAL and AvDL programs. Within the run-time environment these variables are stored in data entries on the bottom of a process's stack, similar to global variables, despite the fact that their scope, which is managed by the language's translator, is not global. This provides the host application with a mechanism to directly affect the execution of running NPC programs.

Embedding the system prototype's virtual machine into a host application using the methods of the run-time environment's API is uncomplicated and just as simple as the integration of the ZBL/0 virtual machine into a game engine (see Chapter 7, Section 7.2.4.2). After the instantiation of the virtual machine by retrieving an instance of the virtual machine object, the minimum requirement for the creation of a virtual entity is the creation of an entity process on the virtual machine, the association of that process with an object that resides in the virtual world and the invocation of the scheduler for every update cycle.

10.4.2 Using the System API

The integration of the prototype system into a host application requires the creation of virtual entities and the instantiation of the virtual machine with which these entities can then be registered, as shown in the following example.

If an entity process is supposed to be associated with an entity object that resides in the host environment, then that object's class must be inherited from the system API's 'entity' class. Any object that is created as an instance of

this derived class is a virtual entity that can be used by the system prototype's virtual machine. If this class requires an entity program to access some of its functionality, the class describing the virtual entity must also inherit from the API's 'callback' class, as shown in the class given below (npc).

```
#include "entity.h"
#include "callback.h"
...
class npc : public svm::entity, public svm::callback
{
    ...
    public:
    bool execute(int method,svm::callstack_t &callstack);
    ...
};
```

For an instance of this virtual entity to be associated with an NPC process, the system's virtual machine needs to be instantiated.

```
#include "vm.h"
...
svm::vm *virtualMachine = NULL; // pointer to the virtual machine
...

{
    // retrieve an instance of the virtual machine
    virtualMachine = svm::vm::Instance();
    ...
```

After the creation of an instance of the virtual machine, the next step is the registration of any events that entity programs need to be notified about.

```
...
```

```
// register the event "event"
virtualMachine->registerEvent("event");
...
```

If there are callback functions that need to be accessible to all entity processes, such as default actions that act as fall-backs (see Section 10.2.1), they should be registered with the virtual machine after the registration of the events. If not, then the next step is to load in the entity programs to create the entities' processes on the virtual machine and then to associate these processes with the entity objects in the host application.

```
...
// create an entity object "gameBot" based on the npc class
npc gameBot;

// create a process that runs the "entity.sbp" program
int proc=virtualMachine->addProgram("entity.sbp");

// associate the entity object with the entity process
virtualMachine->registerEntity(proc,&gameBot);
...
```

Afterwards, any process-specific callback functions, such as those included in the definition of the entity object, need to be registered for the entity process.

```
...
// register the first callback function "cb1"
// (no parameters or return value)
virtualMachine->registerCallback(proc,&gameBot,1,"cb1",false,0);

// register the second callback function "cb2"
// (three parameters and a return value)
virtualMachine->registerCallback(proc,&gameBot,2,"cb2",true,3);
```

```
    ...
}
```

Once the set-up of the entity programs and the virtual machine has been completed, the 'run' method of the run-time environment's API should be called once during every update-cycle of the host application, which is usually once per rendered frame. This method then performs the execution cycle of the virtual machine, first handling all events that have occurred and then executing the entity programs themselves.

```
...
virtualMachine->run();
...
```

The API makes provision for additional operations, such as querying and handling of virtual machine run-time errors, however these are not essential for integrating the system prototype's virtual machine into a host application.

Chapter 11

Analysis of the System

AvDL is a comprehensive scripting language for the definition of virtual entities that populate the virtual worlds of computer games, i.e. NPCs (tactical opponents, incidentals, team-mates and even observers – see Chapter 2, Section 2.2) as well as objects that the NPCs can interact with. The AvDL system provides a synthesis of the functionality of a wide range of different technologies that are used in the development of virtual entities in modern computer games. The system supports different concepts, such as deterministic behaviour as well as goal-oriented behaviour and the means for the creation of annotated entities. All of these are exposed through a consistent language and combined within a single unified system that is generic, i.e. not limited to a single type or genre of computer game. The syntax and structure of AvDL attempts to accommodate novice programmers as well as those who already have some experience with the programming languages C/C++ or Java. Furthermore, in all likelihood due to its similarity to these popular production languages, the system also provides the means for much wider use of AvDL, possibly even in a more generic scripting role.

11.1 Meeting of Criteria

The design of AvDL and its SEAL subset was directed and informed by the requirements that we believe have to be met by a BDL for virtual entities in computer games (see Chapter 5, Section 5.2). In contrast to the ZBL/0 scripting

language (see Chapter 7, Section 7.2) that was our first attempt to create a (procedural) BDL, which does not fully conform to all of these demands, we believe that the requirements that we identified earlier are met by AvDL, and in part also by AvDL's SEAL subset.

11.1.1 Language Requirements

To be useful as a BDL for the creation of virtual entities in computer games of different genres, the language needs to satisfy several criteria (see Chapter 5, Section 5.2.1). One of the requirements for the specification of a BDL was to keep the language generic, a possible solution to which is to base the BDL on an existing production language. Being based on the C++ [Stroustrup 1997] and C [Kerninghan and Ritchie 1988] programming languages, both scripting languages (AvDL and SEAL) fulfil this requirement.

A simple examination of the types of virtual entities found in modern computer games shows that while the generation of simple deterministic behaviour for NPCs is sufficient for some games, other games attempt to increase the believability of their entities by making them display goal-directed behaviour. A generic BDL for use in games will therefore have to accommodate both of these game AI methods. This directly leads to the demand for the inclusion of a state machine data type, as FSMs are one of the most frequently used game AI techniques. AvDL and SEAL both include a state type that allows for simple state machines to be defined using basic instructions and data structures of the language itself without the need for any libraries to extend the capabilities of the system. The provision of goal-orientation specific data types and operators is another BDL requirement that has been met by the AvDL specification.

A further requirement was to keep the BDL simple, i.e. to avoid overloading the language with too many features. This can be achieved by refraining from the use of intrinsic functions that are hard-coded into the BDL's run-time system, and by allowing additional functionality to be provided through an external library of functions. After all, "a language cannot support everything, but conceivably, a large set of libraries could" [Stroustrup 2005]. Unlike the ZBL/0 behaviour definition system, AvDL and SEAL have reduced the system's reliance

on intrinsic functions to a bare minimum, with only a handful of intrinsic functions being integrated with the virtual machine. These functions do not provide any complex operations but are low-level system functions that allow entity programs to directly access the run-time environment. Entity programs can acquire additional functionality through callbacks of the 'action' type, and in the case of AvDL through plug-ins for the system's extension architecture.

A BDL for modern computer games needs to support entity annotation and smart environments, as a behaviour definition system which implements this promising technique provides a powerful means for simplifying the creation of intelligent NPC behaviour. The facilities for this are included with AvDL as well as its SEAL subset and, as its name implies, the SEAL subset of AvDL was especially designed with the creation of annotated virtual entities in mind.

The creation of programs for the definition of virtual entities would benefit from the additional level of abstraction offered by an object oriented approach, which is why simple object orientation is another requirement for the specification of BDLs. AvDL satisfies this demand for object orientation with the inclusion of its 'class' data type, which makes AvDL a language that has all of the features of a BDL, conforming to the requirements that we identified in chapter 5.

The SEAL subset of AvDL may only be a procedural programming language that does not have any object oriented data structures, however, through its FSM and event types it does have game AI specific data types, as well as the operators associated with these types. SEAL can be used to define virtual entities and to annotate them for deployment in smart environments. So, while it is not strictly speaking a BDL according to our definition, as it does not fulfil all criteria, SEAL should still be classified as a BDL, as it satisfies most of them.

11.1.2 Run-Time System Requirements

As the purpose of run-time system's virtual machine is to form the core of a behaviour definition system, it also needs to fulfil certain criteria (see Chapter 5, Section 5.2.2). To allow a host application to be extended with the behaviour definition system, it should be implemented as a separate module that is either embeddable or that can be accessed as a plug-in. By being implemented as a

library that is accessible solely through its API, i.e. an embeddable module, our system prototype does exactly that.

This separation into its own module is a precondition for the requirement to keep the execution of BDL programs apart from the rest of the application. It would be highly undesirable if any mistakes in an entity program, which would result in run-time errors on the virtual machine, could destabilise the host application itself and thus lead to program failure. However, a sufficient degree of run-time stability can be attained by monitoring the execution of entity programs and pre-emptively terminating these if errors that could cause run-time instabilities occur.

The architecture of our prototype's virtual machine maintains the independence of BDL programs from the run-time system's host application. Running entity programs are stopped by the virtual machine when they fail, leading to the graceful degradation of the run-time environment without affecting the running of the host application.

The run-time environment should have as small an overhead as possible for the execution of BDL programs. One way that this aim can be achieved is for the virtual machine to use pre-compiled bytecode rather than to interpret the BDL itself at run-time. This is the exact strategy used in our system prototype's virtual machine that is based on the ZBL/0 virtual machine (see Chapter 7, Section 7.2.3), which has what we believe is a small execution overhead.

The final criterion for the BDL's run-time environment is platform independence, as many games are now simultaneously published for several platforms. This criterion is met by our prototype system, which provides a portable virtual machine that has so far been implemented for the Microsoft Windows operating system.

As a platform independent and robust virtual machine that can be embedded into any C++ based host application, our system prototype fulfils all of the identified requirements for a behaviour definition system's run-time environment.

11.2 Features of the Avatar Description Language

AvDL was developed with the goal of providing a unified method for defining the behaviour of virtual entities in computer games and provides a simple syntactic mechanism for the aforementioned purpose. The design of the language was guided by the aim to make this mechanism accessible to programmers, as well as game designers who may only have a limited knowledge of computer programming. The language is in itself consistent, with different features of the language being exposed to the programmer using similar syntactical elements, e.g. the use of the '@' (at) symbol to precede the definition of weight values for fuzzy states as well as goals (see Chapter 8, Section 8.2.1).

The starting point for the specification of the language was the C++ programming language. A significant modification to C++ was the introduction of the entity type that encapsulates AvDL and SEAL programs, which also provides the BDL program's entry point. This was inspired by the structure of Pascal programs [Wirth 1993] that are encapsulated by the 'program' keyword and the definition of the program's main routine. "A Pascal program has the form of a procedure declaration" [Wirth 1973] that allows the declaration and definition of subroutines, data structures and variables which are used by the program in-between the declaration of the program and the definition of its entry function. This seemed to be a more logical solution for providing the program's entry point than those used by C/C++ and Java, i.e. the requirement to define an entry function with a pre-determined (reserved) identifier, such as "main".

11.2.1 Object Orientation

Object orientation was integrated into the AvDL specification from the very beginning, however, it has been considered less important than several other of the language's features and thus not been given as much attention as the game AI specific data types. The definition of object oriented classes in AvDL is a simplification of object orientation in C++, based on C++ 'struct' records.

Object orientation as such is independent of the virtual machine, i.e. it imposes no special requirements regarding architecture and make-up on the run-time environment. Instead the implementation of object orientation is a compiler issue, as object oriented classes can be broken down into a procedural code representation [Blunden 2002], which in turn can be targeted at the system prototype's instruction set.

11.2.2 FSM Type

The finite state machine type found in AvDL, as well as in SEAL, is probably the most important data type of these languages. FSMs are the cornerstone of most game AI implementations and no BDL would be complete without a means for defining FSMs.

FSMs could have been implemented as a special data structure within the process objects of the system prototype's virtual machine, accompanied by a set of dedicated virtual machine instructions and intrinsic system functions. However, this sort of integration of features would have unnecessarily bloated the virtual machine. Instead, FSMs in AvDL and SEAL work through a combination of the FSM structure's decomposition into basic instructions of the virtual machine and the addition of a switch within each process object that is activated when a state transition is triggered. We believe that this is a more elegant solution than the addition of a separate state data structure to the virtual machine.

11.2.3 FuSM Type

Although the exact makeup and functionality of this data type is not finalised in the AvDL specification, the data type and the semantics described in this thesis (see Chapter 8, Section 8.2.1.3.2) are the most likely candidate for inclusion in the final system. This is a minimalist approach in which the FuSM is only used to maintain the fuzzy states and to manage access to their values. Unlike the system's FSMs, the FuSMs do not have program flow diverted to the state machine. On the one hand the management effort for an FuSM is smaller than that of an FSM, as there are no transitions, which in turn means that for the state machine itself, fewer instructions are needed in the virtual machine. On the

other hand, the burden of providing an interpretation of the FuSM is put on the programmer, which may result in more complex entity programs.

A different fuzzy state interpretation from the above, and one that is frequently applied in game AI development, is that of behaviour competition, which results in the choice of the state that has the highest value or a random selection if that value is simultaneously held by several fuzzy states. Despite the different semantics, the syntax of the above type of FuSM could be utilised virtually unchanged for the behaviour competition FuSM, if an FuSM equivalent to the FSM's 'getstate' operator were added to the language specification. Currently no decision has been reached regarding the final semantics of the FuSM data structure; however, this issue will have to be addressed for the extension of the system towards the implementation of the full AvDL specification (see Chapter 12, Section 12.3).

11.2.4 Goal Data Type

Of all of AvDL's data types the goal data type did evolve the most during the development of the language specification, being the last data type to be finalised apart from the tentative fuzzy state machine type (see Section 11.2.3). Goal-orientation is increasingly employed for the definition of the behaviour of virtual entities in computer games. Earlier versions of the goal type were a lot more deterministic, requiring the programmer to provide each goal with the instructions that would need to be executed to reach the goal, making this type appear like an inverted FSM. This goal type also did not include the provision of weights for goals. This type, which could hardly be called a goal type, would have been far more restrictive than the data structure that is described in this thesis. We believe that the approach presented by Orkin [2004a] provides the most promising solution to goal-orientation in games and the final specification of the goal data type was influenced by this method.

11.2.5 Entity Annotation

Our system prototype provides a working mechanism and interaction protocol that allows entities to be annotated and other entities to utilise these annota-

tions (exported functions). However, the specifications of AvDL and SEAL do not include contingencies for synchronisation problems that may arise if several processes attempt to simultaneously access the same annotated entity. In entity programs this potential problem has been partially addressed by switching off the advertising of exported functions as soon as one process initiates the entity annotation interaction protocol (see Chapter 9, Section 9.3.1), effectively locking access to the annotated entity's exported functions. Depending on the implementation of the entity program this can serve as a simple mechanism for deadlock [Tanenbaum 2001] prevention, as once an entity stops advertising its exported functions no other entity can query the code segments of these functions, effectively denying access to all entities except the one that requested the annotated entity's "silence".

The languages' specifications also do not consider situations in which an exported function attempts to change data in its parent process's memory while being executed from a different process, which could seriously disrupt the working of the system if this occurred in combination with several processes accessing the exported function, as mentioned above.

A solution to these potential problems could take the form of additional safety features that are built into the system's virtual machine, modifications to the language specification or a combination thereof. This issue will have to be addressed by future implementations of the system (see Chapter 12, Section 12.3).

11.3 Concluding Remarks on AvDL and its SEAL Subset

AvDL and SEAL provide a framework for the definition of the behaviour of virtual entities, and while both languages provide data structures that greatly simplify the construction of these virtual entities, their seemingly intelligent behaviour is not generated by the BDLs as such. The purpose of the BDLs is to manage and tie together the functionality of the virtual entities, which is provided to the behaviour definition system by the host application. This means that the

functionality of AvDL and SEAL programs is entirely dependent on the implementation of the virtual entities that they control within their host application. The resulting system is compact and highly extensible. It is our firm belief that the AvDL system matches the requirements for a behaviour definition run-time system, making it sufficient for defining and controlling the majority of artificial entity types that can be found in current computer games.

Chapter 12

Conclusion

12.1 Summary of Contributions

Our exploration of behaviour definition for virtual entities in computer games has yielded several contributions to knowledge.

12.1.1 Syntactic Behaviour Definition for Virtual Entities

The principal contribution of this thesis is the definition of Behaviour Definition Languages (BDLs), illustrated with the design of the AvDL behaviour definition scripting language. AvDL is a new extensible behaviour definition language for virtual entities, developed to conform to our definition of BDLs, which we believe to be suitable for application to NPCs in computer games. Its design was aided by the creation of several other BDLs for virtual entities, namely ZBL/0 and AvDL's SEAL subset, and to a certain degree the GP Asteroids Script language and the C-Sheep mini-language, the development and implementation of which involved an evaluation of different approaches and implementations.

12.1.2 Classification of BDLs and Scripting Systems in Computer Games

Secondary contributions include a comprehensive investigation of scripting languages in computer game development, which led to the proposal of a simple

classification of scripting systems in games. as well as a detailed examination of programming language requirements and design principles in support of our definition of the term behaviour definition language (BDL). For this we also carried out a comprehensive survey of the use of artificial intelligence techniques currently used in computer games [Anderson 2003a] (see also Chapters 2 and 4).

12.1.3 Implementation of a Prototype Behaviour Definition System

In order to test a number of hypotheses relating to the AvDL behaviour definition language we designed and implemented the ZBL/0 scripting system. In particular. we used this system as a test bed for different approaches to the implementation of the interface that enables the exposure of behaviour definition capability to computer game engines, to make the creation of reusable behaviours for virtual entities possible. Progressive refinement of this system's virtual machine has led to the implementation of our system prototype for AvDL's SEAL subset which allows deterministic behaviour definition using FSMs as well as the more emergent behaviour definition that is the result of the use of smart terrain and annotated objects. the procedural definition of which Doyle [2004] in the conclusion of his thesis suggests to be a promising avenue for future investigation.

Our game-genre independent embeddable behaviour definition system exposes different methods of behaviour definition, including the definition of virtual entities, as well as elements of their environment that they can interact with, through a single software interface providing a framework for the creation of NPCs in virtual (game) worlds.

12.2 Discussion

"... language is what gives humans enormous leverage over the universe" [Wilcox 2007]. Analogous to this, scripting languages in games, which provide control over the behaviour of the application, give the programmer "enormous leverage" over the game's virtual reality, and in the case of BDLs, over the virtual entities that inhabit the game world. The aim of our research has been to create a generic

AI behaviour definition system for computer games. which employs a syntactic solution to the problem of behaviour definition. The focus of our work has been to gain a sufficient understanding of the game development process and the use of artificial intelligence techniques in computer games to bring us closer to achieving this goal.

Fuelled by the improvements to graphical realism in games and the growing demand for content to enrich the virtual worlds, the games industry has recently experienced a drive towards automated content generation [Nareyek 2007], which is usually approached using procedural techniques. Syntactic behaviour definition, i.e. the use of a (scripting) language to procedurally program the behaviour of virtual entities, is consequently a step in the right direction. While we are convinced of the usefulness of the BDLs that we created in the course of our work, programming language evaluation is not a trivial task and "often, the choice of programming language comes down to aesthetic issues, which are necessarily subjective", as Horswill [2000] poignantly states.

Our behaviour definition (scripting) system design builds on our understanding of the evolution of scripting languages from the early command-line interpreters to modern embedded systems, as well as common game AI techniques. A common denominator of many of these systems is the need to balance performance (execution speed) and flexibility, which are conflicting objectives. Judging from the performance of its predecessors (ZBL/0 and C-Sheep), which used very similar virtual machines, we believe the execution speed of our system to be adequate for most situations. Furthermore, thanks to the language's extensible and mostly generic nature, which grants it a lot of flexibility, programs that are written in the AvDL behaviour definition scripting language should scale well to the demands of game developers.

> "One characterization of progress in programming languages and tools has been regular increases in abstraction level – or the conceptual size of software designers building blocks" [Garlan and Shaw 1994].

This is reflected in the data structures of AvDL and SEAL, such as the goal and state types that provide bigger "building blocks" for operations that could be decomposed into simpler instructions and constructs of the scripting language,

which would achieve the same effect but require a lot more effort by the programmer and result in a lot more source code. It should be noted that the instruction set of the system prototype's virtual machine was finalised after these language features had been decided upon, however, not all of these features were made accessible through dedicated instructions or intrinsic system functions. Instead, during the design of our prototype system the virtual machine instructions that encode these more abstract data structures were generated by adapting results of the compilation of sequences of the language's regular types and instructions emulating the abstract types. This is achieved by using a modified version of the C-Sheep compiler [Anderson and McLoughlin 2006], which targets a predecessor of the system prototype's virtual machine that shares a large proportion of its instruction set with the system prototype's virtual machine. This means that various game AI specific data structures of the language are effectively mapped to the existing instruction set of the virtual machine. In hindsight, the decision not to implement AvDL's features by simply adding appropriate data structures to the virtual machine and exposing access to them through special virtual machine instructions was arguably the most important choice regarding the architecture of the virtual machine. Many of the instructions incorporated into the virtual machine closely mirror machine instructions that are built into existing hardware, which means that in the future it may be possible to compile AvDL programs into native code using a true OTF compiler (see Section 12.3.2), which in turn could boost the performance of the behaviour definition system. This. however, would not be a possibility if a different approach to the design of the virtual machine and its instruction set had been followed.

We believe that our system prototype has all of the characteristics of a modern scripting system for game development and that, although not yet complete, it is already usable. However, ultimately a field trial may need to be conducted when the first full prototype that completely implements the AvDL specification is ready for deployment to verify the suitability of the language for computer game development.

12.3 Future Work

A by-product of our exploration of behaviour definition systems is the C-Sheep project, a system for the teaching of introductory computer science and programming which is work in progress and continues to evolve. Future versions of the C-Sheep system should benefit from some of the conclusions arrived at during the course of our research and the work presented in this thesis. This work itself is continually evolving and we endeavour to improve the system's prototype, as we believe that additional work is necessary before the system is mature enough for being incorporated into a game engine.

An obvious starting point is the extension of our system prototype to implement the full AvDL specification. In addition to this we have already begun to implement a recursive descent compiler for AvDL's SEAL subset which we plan to integrate with the system and which will eventually lead to the provision of a full compiler for AvDL programs that implements all features of the AvDL specification. The integration of this compiler into the run-time system first as an AOT compiler for the compilation of entity programs at process initialisation within the virtual machine, and then as a type of OTF compiler for the replacement of existing functions during program run-time, possibly allowing the creation of self-modifying virtual entities, would bring the system closer to completion and increase its usefulness for game development.

In addition to the completion of the system we have also identified several lines of investigation which we intend to follow.

12.3.1 Language Additions

While the AvDL language includes all of the features that we consider necessary for BDLs, the system could benefit from the exploration of the integration of additional features, such as history-augmented inputs as described by Blow [2002], which would allow AvDL programs to provide virtual entities with a memory of their actions. These would be very useful for implementing machine learning functionality, greatly increasing the flexibility of the system.

A further useful technique for game AI behaviour definition, which we plan to evaluate for possible inclusion into the AvDL system, is messaging [Rabin 2000b].

This would allow communication between virtual entities beyond the capabilities of the current system which relies on the triggering of simple events as its only means of communication.

We are also considering to investigate the feasibility of an expansion of the AvDL specification to support "plot scripts" (as opposed to conventional programs) to allow for the better application of our system to story-driven computer games. This kind of script is often found in interactive drama [Magerko 2006] – a new genre of digital entertainment – where it is used to author the plot of the game.

In addition to the possible introduction of these new language features, we also plan to review the effectiveness of the current specification of the language. Whereas a certain degree of redundancy in the definition of a programming language undoubtedly increases the flexibility of the language by allowing various tasks to be carried out using different methods, as is the case with AvDL (see appendix D), an unwanted side effect can be that this may make it a lot more complicated to read and to understand programs that have been written in this language. Consequently it would be prudent to re-evaluate some of AvDL's duplicate features after field trials of the system have been conducted and to streamline the language specification by removing or redesigning features that cause unnecessary confusion.

Finally, we plan to complete the language specification with an expansion of the standard function library for the AvDL scripting language, possibly through the use of a plug-in for the system's extension architecture. In addition to the functions for the use of annotated entities, this AvDL standard library should provide standard functions and define appropriate compound data types for solving a variety of game AI tasks. These functions should ideally adhere to the standards suggested by the IGDA AI Interface Standards Committee [Nareyek et al. 2004]. This should then allow the generation of AI entities for a wide range of different computer games, making AvDL a truly generic behaviour definition system for NPCs.

12.3.2 Run-Time System

Apart from general optimisation of the system's virtual machine, a useful extension to the run-time system would be the addition of support for run-time debugging.

A further direction that should be explored is the virtual machine architecture itself. While stack architectures are the most common virtual machine architecture [Davis et al. 2003], a register based approach may be more efficient [Shi et al. 2005] and also provide more opportunities for bytecode optimisation. While this would also have implications on code generation for the system's compiler, those would be minor, as code generation for stack architectures can be adapted for register allocation with relatively little effort [Wirth 1996].

As the prototype system's instruction set resembles the instructions found in real microprocessors, the virtual machine could also be enhanced by the integration of an OTF compiler for translating virtual machine instructions into native machine code, which should provide a significant increase to the run-time system's performance.

12.3.3 System API

Linked to the improvements to the run-time system are enhancements to the system's API. The AvDL API is an integral part of the AvDL virtual machine as it provides the interface that allows host applications to access the virtual machine and in turn provides AvDL programs running on the virtual machine with access to data and functions that are defined within the host application.

The versatility of the behaviour definition system could be substantially improved by allowing access to plug-ins of the system's virtual machine through the API. As a side effect, this could provide a partial solution to the definition of a plug-in based standard library for the AvDL system, considering that the host application would probably require a means for providing information to this library.

A final avenue for exploration is the implementation of the BDL's 'action' type for which our system prototype employs callback objects, an approach that may be too inflexible for computer game development. An alternative function

binding method could involve the use of functors or possibly a template-based approach.

Appendices

Appendix A

A* Sample Implementation

Chapter 4 examined common approaches to the implementation of AI in computer games. This appendix presents an implementation of the A* algorithm, which is the most frequently used path planning algorithm in game development (see Chapter 2, Section 2.3.2.1 and Chapter 4, Section 4.3). Here it is implemented as a C++ function, utilising the C++ STL (Standard Template Library).

A.1 Dependencies

The presented implementation depends on the definition of several data structures and functions. These are a node structure, containing positional information (see Chapter 4, Section 4.3.2), a pathnode structure, extending the node by heuristic information used by A* (see Chapter 4, Section 4.3.3), and a function for estimating the cost of travel (see Chapter 4, Section 4.3.2). For the reader's convenience these dependencies are repeated below:

A.1.1 Node

```
struct node
{
  double x;
  double y;
  double z;
```

```
  double p;
  struct node **neighbours;
};
```

A.1.2 Pathnode

```
struct pathnode
{
  node  *mapnode;    // node within the graph
  double fitness;    // sum of the goal and heuristic values
  double goal;       // cost of travel up to current node
  double heuristic; // estimated cost of travel to destination
  pathnode *parent; // parent node within the path
};
```

A.1.3 Cost of Travel

```
double cost(node *s,node *d)
{
    double h = 1.0;
    double x = (d->x - s->x)*(d->x - s->x);
    double y = (d->y - s->y)*(d->y - s->y);
    double z = (d->z - s->z)*(d->z - s->z);
    double c = sqrt(x+y+z);
    c *= (s->p+d->p)/2.0;
    h += (d->y - s->y) / (5.0*fabs(d->y - s->y));
    c *= h ;
    return c;
}
```

A.2 A* Function

Given the above node data structures and cost function (see Section A.1 above), a function implementing A* path planning could be written as shown below:

```
std::list<node*> aStar(node *start, node *goal)
{
   std::list<pathnode*> open_list;
   std::list<pathnode*> closed_list;
   pathnode *P=new pathnode; // start node
   P->mapnode=start;
   P->goal=0;
   P->heuristic= cost(start,goal);
   P->fitness=P.goal+P.heuristic;
   P->parent=NULL;
   open_list.push_back(P); // add start node to open list
   while(!open_list.empty())
   {
      std::list<pathnode*>::iterator n=open_list.begin();
      pathnode *B=n; // best node in open list
      for(int i=0;i<(int)open_list.size();i++,n++)
      {
         if(n->fitness<B->fitness) B=n; // select best node
      }
      open_list.remove(B); // remove best node from open list
      if(B->mapnode==goal) // success - path discovered
      {
         std::list<node*>path;
         while(B!=NULL) // construct path by retracing to start
         {
            path.push_front(B->mapnode);
            B=B->parent;
         }
```

```
while(!open_list.empty()) // clean up
{
    pathnode *temp=open_list.front();
    delete temp;
    open_list.pop_front();
    }
    while(!closed_list.empty()) // clean up
    {
        pathnode *temp=closed_list.front();
        delete temp;
        closed_list.pop_front();
    }
    return path;
}
// now process all of the best node's links
for(int i=0; B->mapnode->neighbours[i]!=NULL;i++)
{
node *Cmapnode=B->mapnode->neighbours[i];
double g=B->goal+cost(B->mapnode,Cmapnode);
double h=cost(Cmapnode,goal);
double f=g+h;
bool foundC=false;
n=open_list.begin();
for(int j=0;j<(int)open_list.size();j++,n++)
{
    if(n->mapnode==Cmapnode) // if in open list
    {
        foundC=true;
        if(g<n->goal) // better path found
{
    n->goal=g;
    n->heuristic=h;
    n->fitness=f;
```

```
                n->parent=B;
            }
        }
    }
    if(!foundC)
    {
        n=closed_list.begin();
        for(int j=0;j<(int)closed_list.size();j++,n++)
        {
            if(n->mapnode==Cmapnode) // if in open list
            {
                foundC=true;
                if(g<n->goal) // better path found
                {
                    n->goal=g;
                    n->heuristic=h;
                    n->fitness=f;
                    n->parent=B;
                }
            }
        }
    }
    if(!foundC) // C has never been processed
    {
        pathnode *C=new pathnode;
        C.mapnode=Cmapnode;
        C.goal=g;
        C.heuristic=h;
        C.fitness=f;
        C.parent=B;
        open_list.push_back(C); // add C to open list
    }
}
```

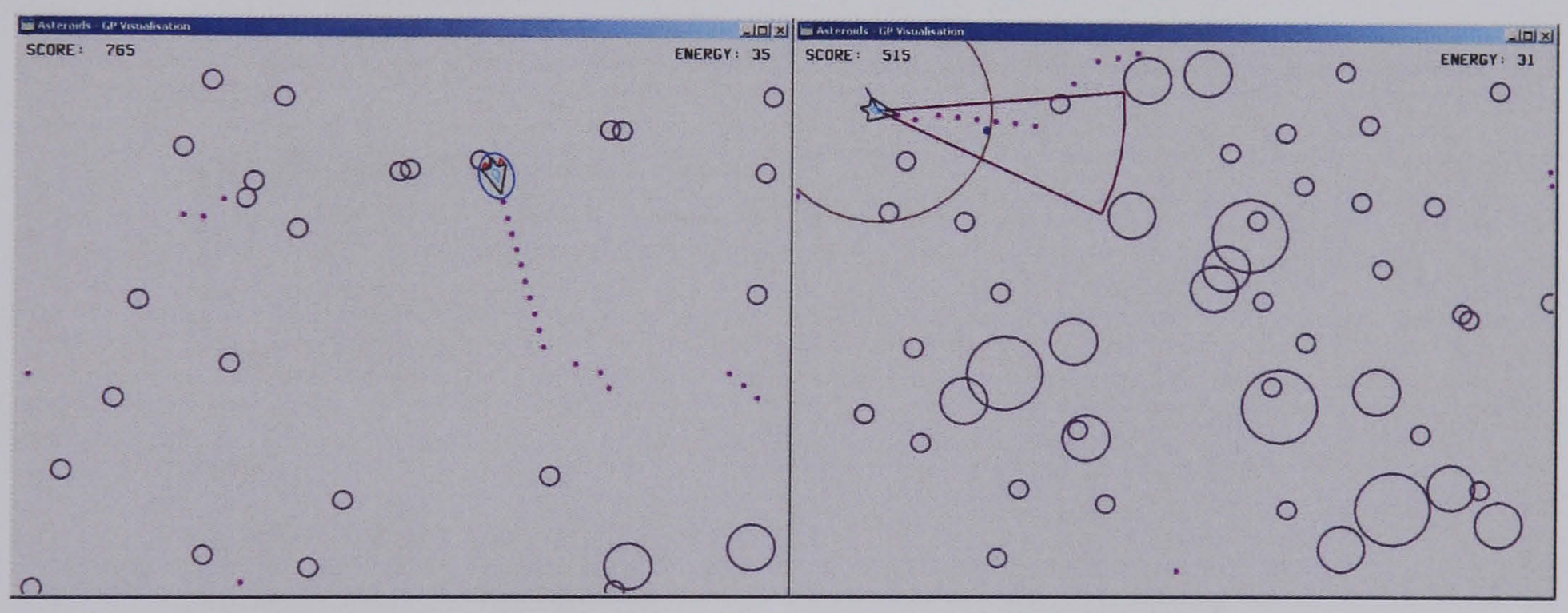
```
        closed_list.push_back(B); // add node B to closed list
    }
    while(!closed_list.empty()) // clean up
    {
        pathnode *temp=closed_list.front();
        delete temp;
        closed_list.pop_front();
    }
    return std::list<node*>(); // no path could be found
}
```

The data returned by the above function is a (C++ STL) list storing the nodes of the path from start node to destination node for use by the NPC.

Appendix B

GP Asteroids Script

This appendix presents the syntax and functions of the GP Asteroids Script language discussed in Chapter 7 (Section 7.1) of this thesis, which we developed in our investigation of genetic programming for NPC behaviour definition [Anderson 2002].



GP Asteroids Script is a LISP like scripting language for the definition of player behaviour for virtual entities competing in the classic arcade game Asteroids.

There are three versions of the scripting language.

1. A basic version.

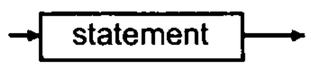
2. A version with additional "automatically defined functions" (ADFs) that

also incorporates several syntactic constraints on the program's "result producing branch" (RPB).

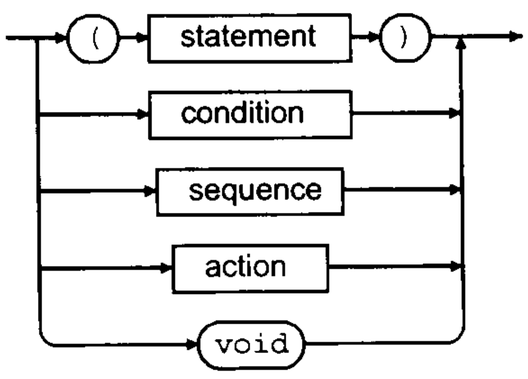
3. A version that adds several "super actions" to the basic version of the language.

B.1 Original Language Definition

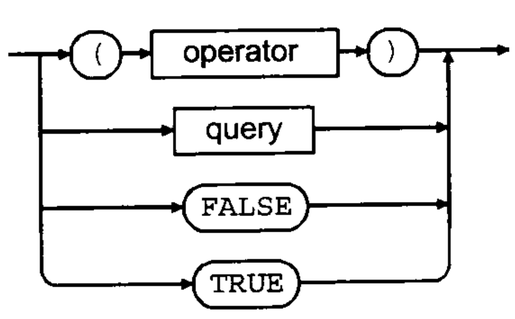
program:



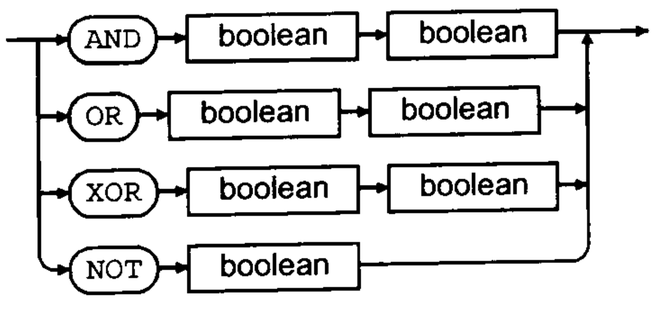
statement:



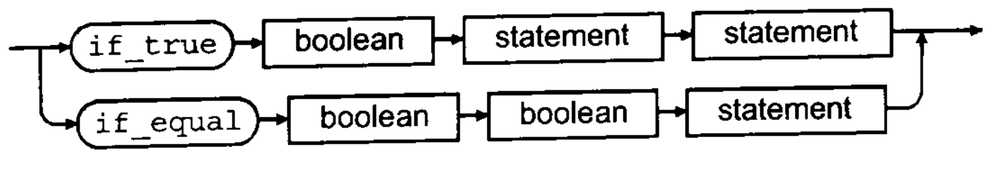
boolean:



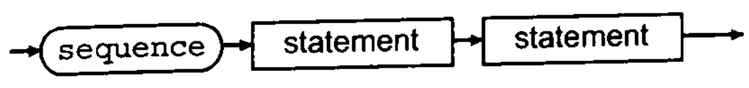
operator:



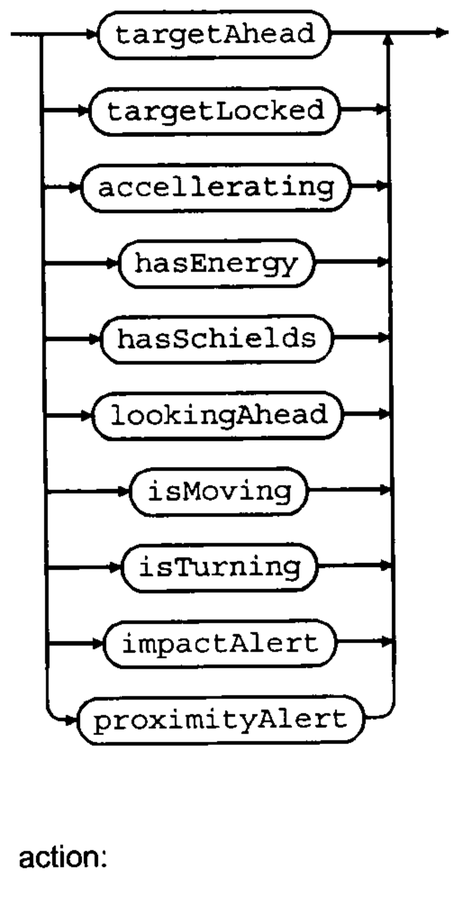
condition:



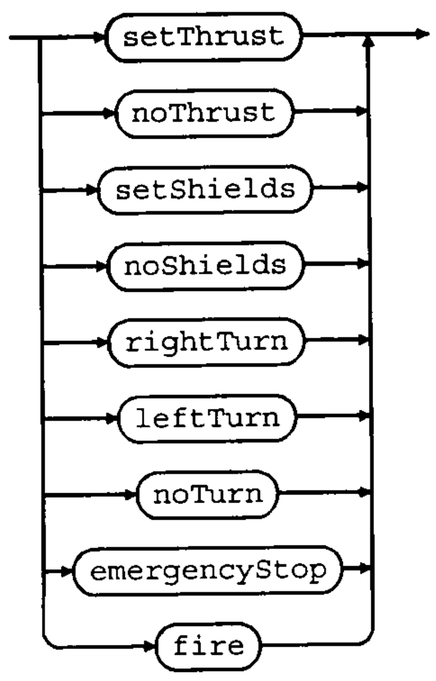
sequence:



query:

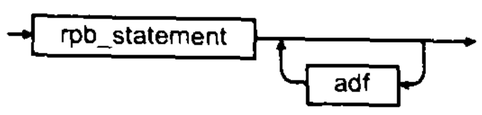


action:

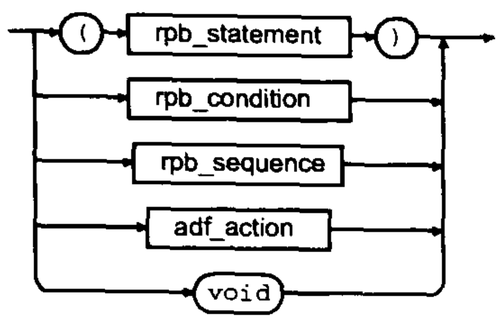


B.2 GP Asteroids Script with ADFs

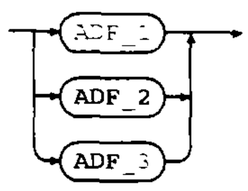
program:



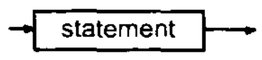
rpb_statement:



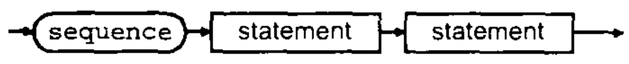
adf_action:



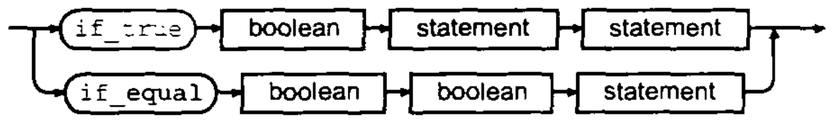
adf:



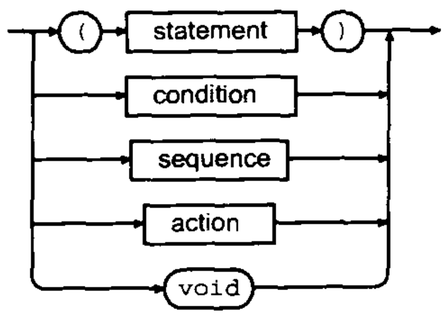
sequence:



condition:

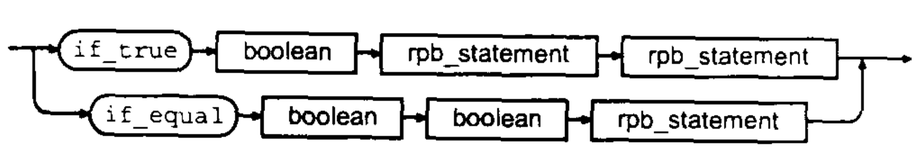


statement:

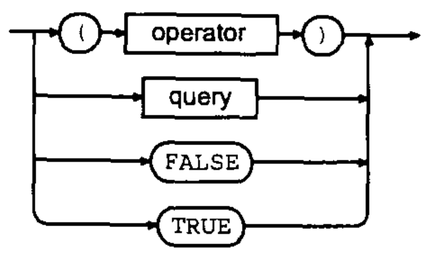


rpb_sequence:

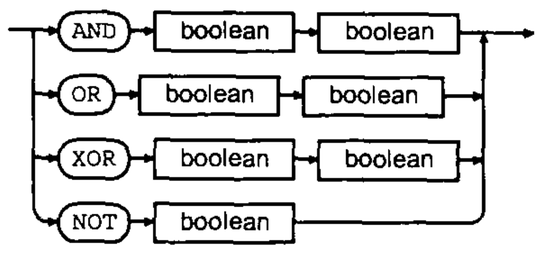
rpb_condition:



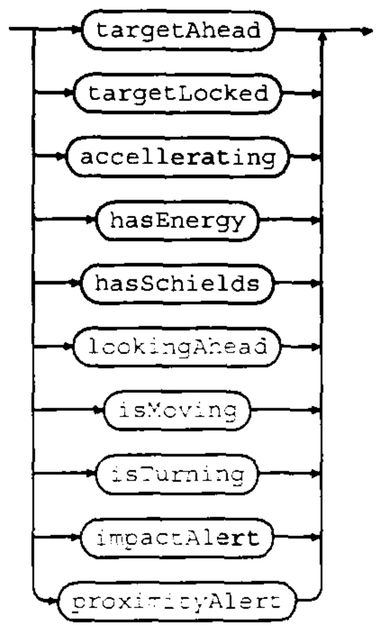
boolean:



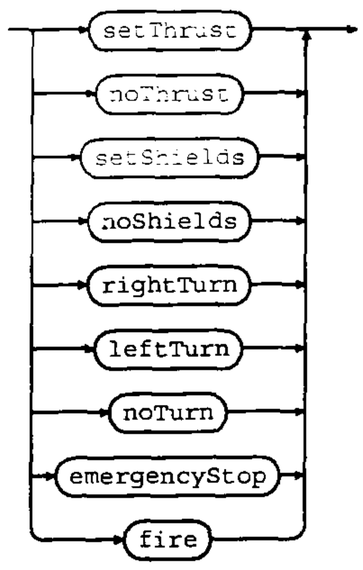
operator:



query:



action:

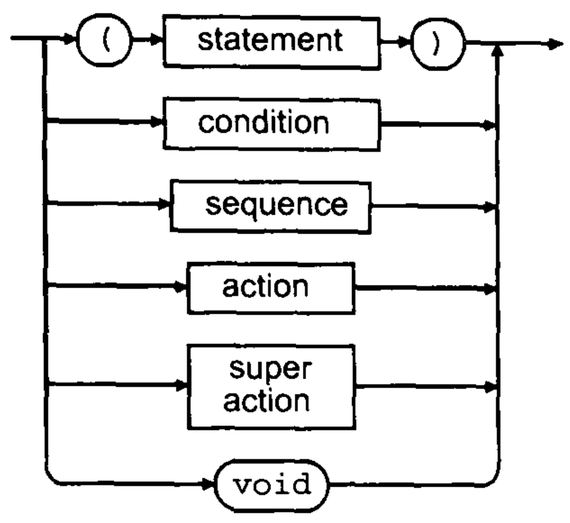


note: "ADF_1", "ADF_2" and "ADF_3" are identifiers for calling each of the three possible automatically defined functions in GP Asteroids Script programs.

B.3 GP Asteroids Script with Super Actions

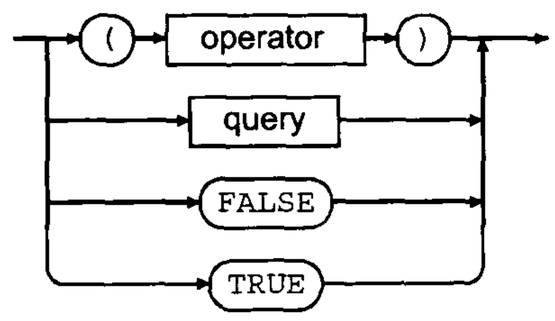
program:

statement:

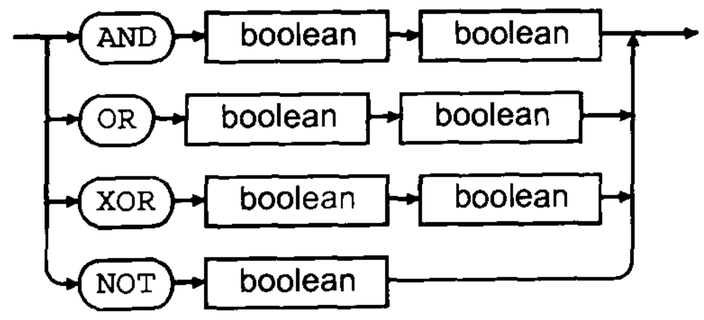


sequence:

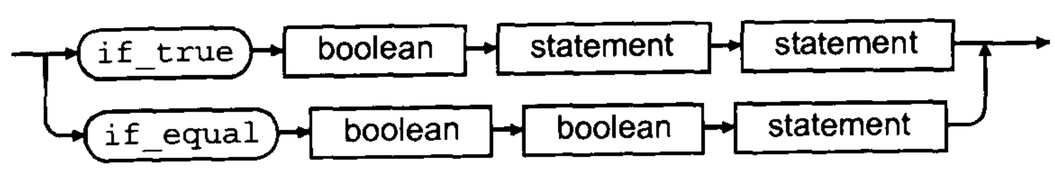
boolean:



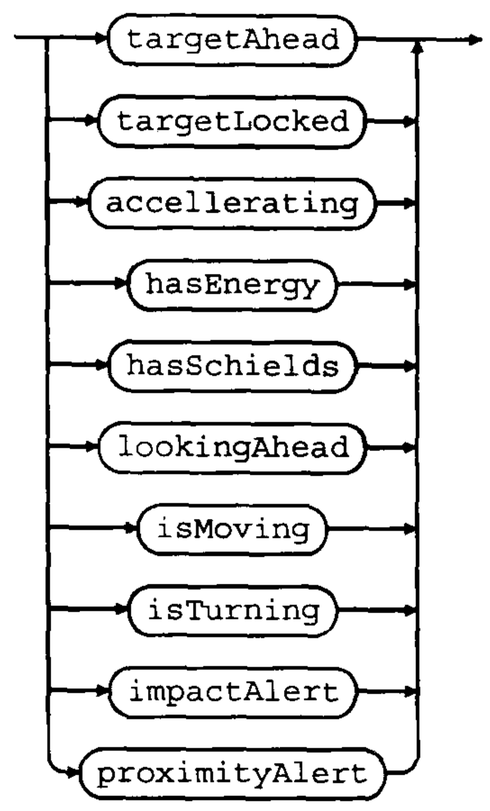
operator:



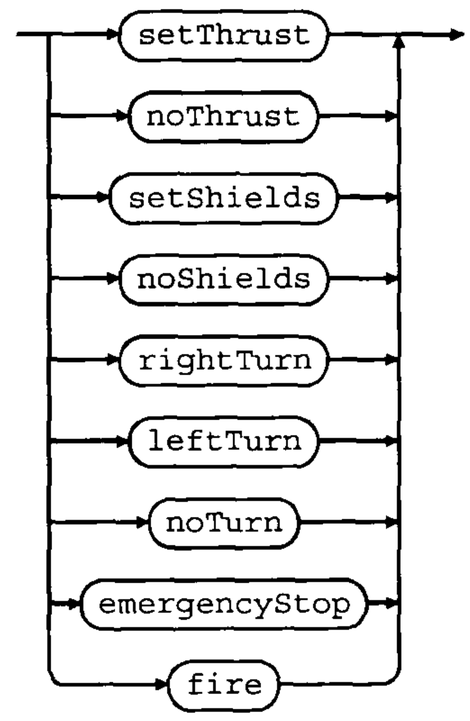
condition:



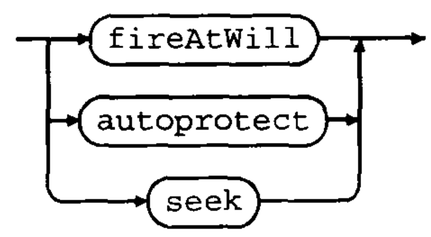
query:



action:



super-action:



B.4 GP Asteroids Script Functions

B.4.1 Sensor Functions

The language's sensor functions all return a Boolean value.

(targetAhead)
TRUE if an asteroid is within the player's field of view, else FALSE.

(targetLocked)
TRUE if an asteroid is in the player's direct line of fire, else FALSE.

(proximityAlert)
TRUE if an asteroid is in the player's proximity (within 12 units from the player), else FALSE.

(impactAlert)
TRUE if the player is about to collide with an asteroid (asteroid is within 3 units from the player), else FALSE.

(hasEnergy)
TRUE if the player has energy left, else FALSE.

(plentyEnergy)
TRUE if the player has enough energy for firing more than four shots, else FALSE.

(hasShields)
TRUE if the player's shields are raised, else FALSE.

(lookingAhead)
TRUE if the player's direction of movement is identical to the player's heading, else FALSE.

(isMoving)
TRUE if the player is moving, else FALSE.

(accelerating)
TRUE if the player has active thrusters, else FALSE.

(isTurning)
TRUE if the player is turning, else FALSE.

B.4.2 Action Functions

Action functions are commands (procedures) that do not return a value.

(setThrust)
Activates the player entity's thrusters.

(noThrust)
Deactivates the player entity's thrusters.

(decelerate)
Reduces the player entity's speed.

(setShields)
Raises the player entity's shields.

(noShields)
Lowers the player entity's shields.

(rightTurn)
Sets the player entity to turn clockwise.

(leftTurn)
Sets the player entity to turn anti-clockwise.

(noTurn)
Sets the player entity to stop turning.

(fire)
Fires a single projectile.

B.4.2.1 Super Actions

(autoprotect)
Automatically raises the player entity's shields and activates its thrusters if an asteroid gets too close.

(seek)
Moves the player entity across the screen in search of a target.

(fireAtWill)
Makes the player entity automatically fire projectiles at asteroids that are in its line of fire.

B.4.3 Control Structures

The language's control structures direct program flow.

(if_true `b v1 v2`)
If the Boolean function b returns TRUE the void procedure $v1$ is executed, else if b returns FALSE the void procedure $v2$ is executed.

(if_equal `b1 b2 v`)
If the return values of the Boolean functions $b1$ and $b2$ are identical the void

procedure v is executed.

(sequence v1 v2)

Executes the two void (action) functions $v1$ and $v2$ one after the other.

Appendix C

The ZBL/0 Programming Language

This appendix describes in detail the ZBL/0 behaviour definition system that we developed for the game engine described in Zerbst et al [2003], as discussed in Chapter 7 (Section 7.2) of this thesis.

C.1 Game-Bot Scripting Language

ZBL/0 is a simple scripting language for the definition of artificial behaviour for virtual entities in computer games (game-bots). There is only one variable data type in ZBL/0 which can be used to store integer values as well as floating point values. The functions for controlling game-bots are intrinsic to the ZBL/0 scripting language, i.e. they are built into the language and do not have to be activated by means of inclusion of a standard library of functions. Functions can be user-defined, but function parameters in user-defined functions are not supported by the language and have to be emulated through the use of global variables. The command syntax of ZBL/0 is similar to that of related procedural languages such as C by Kernighan and Ritchie [1988], Pascal by Wirth [1993] or PL/0 by Wirth [1986]. Each instruction in ZBL/0 is terminated with a semicolon (';'). Programs in ZBL/0 are terminated with a full stop ('.'). The ZBL/0 language is not case-sensitive.

C.1.1 Core Functionality

const	do	else	function	if
return	then	uses	var	while

Table C.1: ZBL/0 reserved words.

ZBL/0 has a very small set of core instructions for implementing structural program elements and functions (see Table C.1).

LOADING OF EXTENSIONS:

Extension libraries are loaded at the start of the program above all declarations using the 'uses' keyword, followed by one or more identifiers (separated by commas) that must match the name of the extension(s).

```
uses <identifier>
```

or

```
uses <identifier>,<identifier>
```

Example:

```
uses printLib;
```

COMMENTS:

The ZBL/0 scripting language has only line-comments, i.e. there are no block-comments. Line-comments are marked with the '#' (hash) character (ASCII 35). Any character following the line-comment symbol until the next new line (symbol) will be ignored by the compiler.

OPERATORS:

The operators available in ZBL/0 are standard arithmetic and logical operators (see Table C.2).

Priority	Symbol	Description
1	!	logical negation
2	^	raise to power
3	/	division
	*	multiplication
	%	modulo
4	+	addition
	-	subtraction
5	=	equality comparison
	<>	non-equality comparison
	<	less-than comparison
	<=	less-or-equal comparison
	>	more-than comarison
	>=	more-or-equal comparison
6	&	logical AND
	|	logical OR
7	=	value assignment

Table C.2: ZBL/0 operator precedence.

BLOCKS:

Blocks of statements can be created by inserting statements between opening braces ('{') and closing braces ('}'). For blocks that enclose functions, constants, variables and functions with a local scope can be defined above the block. Blocks themselves are treated like statements and must be followed by a semicolon.

Example:

```
{
   statement1;
   statement2;
   statement3;
};
```

CONSTANT DEFINITION:

Constants are defined above a block or globally using the 'const' keyword, followed by one or more constants (separated by commas).

```
const <identifier> = <value>;
```

or

```
const <identifier> = <value>, <identifier> = <value>;
```

Example:

```
const false = 0, true = 1;
```

VARIABLE DECLARATION AND DEFINITION:

Variables are declared (and can be defined) above a block or globally using the 'var' keyword, followed by one or more variables (separated by commas).

```
var <identifier>;
```

or

```
var <identifier> = <value>;
```

or

```
var <identifier>, <identifier>;
```

or

```
var <identifier> = <value>, <identifier> = <value>;
```

Example:

```
var x = 10, y;
```

FUNCTION DEFINITION:

Functions are defined as a block of statements following the 'function' keyword.

```
function <identifier>; <block>;
```

Example:

```
function turn_away;
{
   turn_left;
   turn_left;
```

```
};
```

By default the value returned by functions is 1 (one). If any other value is to be returned this has to be done using the 'return' statement.

```
return;
```

or

```
return <value>;
```

Example:

```
function false;
{
   return 0;
};
```

Using the 'return' statement will exit the function. If no return value is provided, the default return value 1 (one) will be returned.

Example:

```
function true;
{
   return;
};
```

Functions are called by using the function's identifier (name) followed by a semicolon. Recursions are possible. Functions can be nested, so the definition of local functions with a limited scope is possible.

CONDITIONAL STATEMENTS:

There is one conditional statement in the ZBL/0 scripting language which allows for one optional alternative. The statement consists of the 'if' keyword followed by a conditional expression and the 'then' keyword followed by a statement. Alternatives can be expressed by following the above with the 'else' keyword followed by the alternative statement.

```
if <expression> then <statement>;
```

or

```
if <expression> then <statement> else <statement>;
```

ITERATIONS (LOOPS):

The only iterative construct in the ZBL/0 scripting language is the head-controlled loop. The statement consists of the 'while' keyword followed by a conditional expression and the 'do' keyword followed by a block.

```
while <expression> do <block>;
```

Example:

```
# a simulated for loop
i=0;
while i<10 do
{
   i=i+1;
};
```

C.1.2 ZBL/0 Function Set

ZBL/0 has a very small set of powerful (partially context-sensitive) core instructions. This section describes these ZBL/0 standard functions and their usage. Each ZBL/0 function is presented together with the corresponding function prototype of the ZBL-API's game-bot interface.

HOUSEKEEPING FUNCTIONS:

The language's housekeeping functions include instructions that directly control the existence of a game-bot within the virtual world, as well as receptors that provide world-state information that cannot be perceived by the game-bot's regular sensors.

ZBL/0: `danger`

ZBL-API: `int zb_checkDanger(void);`

Returns the value 1 if an enemy entity is close to the game-bot (sets an internal game-bot state).

ZBL/0: `die`
ZBL-API: `void zb_doDie(void);`
Kills the game-bot.

ZBL/0: `find <object>`
ZBL-API: `void zb_doFind(int);`
Makes the game-bot follow a path to the memorized location of <object> (see 'memorize') - should set an internal "path-following" game-bot state.

ZBL/0: `idle`
ZBL-API: `int zb_checkIdle(void);`
Returns the value 1 if no path is being followed and there is no danger (sets an internal game-bot state).

ZBL/0: `initialize <xpos>, <ypos> [, <zpos>]`
ZBL-API: `void zb_doInitialize(double,double,double);`
Initialises the game-bot at the given coordinates (in 2D or optionally 3D) in the virtual world (dependent on the implementation of the host application). This function should be the first function invoked by a game-bot after program start (see also function 'spawn').

ZBL/0: `memorize <object>`
ZBL-API: `void zb_doMemorize(int);`
Memorises the location of an <object> (should be stored in a location list) – for use with the 'find' function.

ZBL/0: `respawn`
ZBL-API: `void zb_doRespawn(void);`
Resets the game-bot program and restarts it from its beginning.

ZBL/0: `spawn`
ZBL-API: `void zb_doSpawn(void);`
Initialises the game-bot at random level co-ordinates (implementation dependent). Can be used as an alternative to the function 'initialize'.

ZBL/0: `spawned`
ZBL-API: `int zb_checkSpawned(void);`
Returns the number of respawns of the game-bot.

MODIFIER FUNCTIONS:

The invocation of several ZBL/0 functions requires the provision of additional information to direct their run-time behaviour. This is achieved using so-called modifiers, which are functions that alter the behaviour of other functions that they are combined with.

ZBL/0: `back`
ZBL-API: `int zb_mdfBack(void);`
For use with the 'blocked' function – returns the value 1 if true (path blocked to the back), or 0 if false.

ZBL/0: `front`
ZBL-API: `int zb_mdfFront(void);`
For use with the 'blocked' function – returns the value 1 if true (path blocked to the front), or 0 if false.

ZBL/0: `left`
ZBL-API: `int zb_mdfLeft(void);`
For use with the 'blocked' function – returns the value 1 if true (path blocked to the left), or 0 if false.

ZBL/0: `object`
ZBL-API: `int zb_mdfObject(void);`
For use with the 'face' function – selects the closest object to the game-bot and returns the object's ID.

ZBL/0: `right`
ZBL-API: `int zb_mdfRight(void);`
For use with the 'blocked' function – returns the value 1 if true (path blocked to the right), or 0 if false.

ZBL/0: `target`
ZBL-API: `int zb_mdfTarget(void);`
For use with the 'face' function – selects the closest target entity to the game-bot and returns the target's ID.

CONTROL-FUNCTIONS:

Control functions allow the game-bot to navigate the virtual world and to interact with its environment. As some of these actions might not execute immediately, such as animations that are performed over a specific duration, the execution of the game-bot process may have to be halted for that duration. This is achieved by calling the game-bot interface's 'zb_setBusy' method within the implementation of the game-bot in the host application. It is imperative that busy processes are resumed after the execution of the action has finished. This is done using the method 'zb_unSetBusy'.

ZBL/0: `backstep`
ZBL-API: `void zb_doBackstep(void);`
Makes the game-bot move 1 unit backward.

ZBL/0: `crawl`
ZBL-API: `void zb_doCrawl(void);`
Makes the game-bot move 1 unit forward while ducked.

ZBL/0: `duck`
ZBL-API: `void zb_doDuck(void);`
Makes the game-bot duck down.

ZBL/0: `face <modifier>`
ZBL-API: `int zb_doFace(int);`
Turns the game-bot towards the selected <modifier> (target or object) – returns the ID returned by the modifier function.

ZBL/0: `fire`
ZBL-API: `void zb_doFire(void);`

Makes the game-bot fire the currently selected weapon.

ZBL/0: `jump`
ZBL-API: `void zb_doJump(void);`
Makes the game-bot jump 1 unit forward.

ZBL/0: `jump_back`
ZBL-API: `void zb_doJumpBack(void);`
Makes the game-bot jump 1 unit back.

ZBL/0: `jump_left`
ZBL-API: `void zb_doJumpLeft(void);`
Makes the game-bot jump 1 unit to the left.

ZBL/0: `jump_right`
ZBL-API: `void zb_doJumpRight(void);`
Makes the game-bot jump 1 unit to the right.

ZBL/0: `jump_up`
ZBL-API: `void zb_doJumpUp(void);`
Makes the game-bot jump up on the spot.

ZBL/0: `step`
ZBL-API: `void zb_doStep(void);`
Makes the game-bot move 1 unit forward.

ZBL/0: `strafe_left`
ZBL-API: `void zb_doStrafeLeft(void);`
Makes the game-bot move 1 unit to the left.

ZBL/0: `strafe_right`
ZBL-API: `void zb_doStrafeRight(void);`
Makes the game-bot move 1 unit to the right.

ZBL/0: `turn <angle>`
ZBL-API: `void zb_doTurn(double);`
Turn the game-bot by <angle> degrees.

ZBL/0: `turn_left`
ZBL-API: `void zb_doTurnLeft(void);`
Turns the game-bot by 90 degrees to the left (counter-clockwise).

ZBL/0: `turn_right`
ZBL-API: `void zb_doTurnRight(void);`
Turn the game-bot by 90 degrees to the right (clockwise).

ZBL/0: `use <object>`
ZBL-API: `void zb_doUse(int);`
Make <object> the currently selected object of the game-bot.

SENSOR-FUNCTIONS:

Sensor functions are used to provide a game-bot with information about itself and its environment. This information allows the game-bots to navigate and act in the virtual world.

ZBL/0: `alive`
ZBL-API: `int zb_checkAlive(void);`
Returns the value 1 if the game-bot is alive.

ZBL/0: `armour`
ZBL-API: `double zb_checkArmour(void);`
Returns the armour of the game-bot.

ZBL/0: `blocked <modifier>`
ZBL-API: `int zb_checkBlocked(int);`
Returns the return value of the <modifier> function.

ZBL/0: `health`

ZBL-API: `double zb_checkHealth(void);`
Returns the health of the game-bot.

ZBL/0: `object_ahead`
ZBL-API: `int zb_checkObjectAhead(void);`
Returns the value 1 if the game-bot is facing an object.

ZBL/0: `obstacle`
ZBL-API: `int zb_checkObstacle(void);`
Returns the ID of an obstacle directly in front of the game-bot or the value 0 if no obstacle is directly in front of the game-bot.

ZBL/0: `owns <object>`
ZBL-API: `int zb_checkOwns(int);`
Returns the value 1 if the game-bot has <object> in its inventory.

ZBL/0: `target_ahead`
ZBL-API: `int zb_checkTargetAhead(void);`
Returns the value 1 if the game-bot is facing a target entity.

ZBL/0: `target_alive`
ZBL-API: `int zb_checkTargetAlive(void);`
Returns the value 1 if the nearest detectable target entity is alive.

ZBL/0: `target_armour`
ZBL-API: `double zb_checkTargetArmour(void);`
Returns the armour (value) of the nearest target entity.

ZBL/0: `target_health`
ZBL-API: `double zb_checkTargetHealth(void);`
Returns the health (value) of the nearest target entity.

ZBL/0: `using`
ZBL-API: `int zb_checkUsing(void);`

Returns the ID of the object currently used by the game-bot.

OTHER FUNCTIONS:

ZBL/0: `rnd <limit>`
This function is defined within the virtual machine itself, so there is no corresponding function for the game-bot interface of the ZBL-API. The function returns a random number between the values 0 and the given <limit>.

C.2 Virtual Machine Interface of the ZBL-API

The integration of game-bots into host applications is enabled by the ZBL/0 virtual machine which is controlled by the methods of the virtual machine interface of the ZBL-API.

```
double zbl_getVersion(void);
char *zbl_getVersionString(void);
```
These methods return version information about the virtual machine as a version number or a string (that holds the version number) respectively. This can be used to verify compatibility between the virtual machine and compiled ZBL/0 programs.

```
int zbl_addProcess(char *filename,zblbot *bot);
```
This method adds a game-bot to the ZBL/0 virtual machine. For this it receives two parameters, the first of which is the file name of a compiled game-bot program and the second is the memory address of the game-bot object (derived from the game-bot interface class) representing the entity that is to be controlled by the program. The 'zbl_addProcess' method returns the process ID of the newly added game-bot process.

```
void zbl_removeProcess(int pID);
```
This method removes the game-bot process with the given process ID from the virtual machine.

```
void zbl_replaceProcess(int pID,char *filename);
```
This method replaces the program running in the game-bot process with the given process ID with the compiled game-bot program whose filename is given (see also method 'zbl_addProcess').

```
void zbl_replaceBot(int pID,zblbot *bot);
```
This method replaces the game-bot object associated with the given game-bot process ID with the given game-bot object (see also method 'zbl_addProcess').

```
void zbl_resetProcess(int pID);
```
This method resets the game-bot process with the given ID to its initial state and restarts the game-bot program. The method's effect is similar to that of an invocation of the ZBL/0 function 'respawn'.

```
void zbl_setPriority(int pID,int priority);
```
This method sets the execution priority for the game-bot process with the given process ID to the given value.

```
int zbl_getPriority(int pID);
```
This method returns the execution priority of the game-bot process with the given process ID.

```
void zbl_getExtensions(int pID);
```
This method returns the number of extensions loaded by the game-bot process with the given process ID.

```
int zbl_getActiveProcesses(int pID);
```
This method returns the number of active game-bot processes in the virtual machine.

```
int zbl_run(void);
```
This method is the most important method of the ZBL-API's virtual machine interface. It incorporates the virtual machine's scheduling mechanism for the ex-

ecution of game-bot processes. The 'zbl_run' method must be invoked once for every update-cycle of the host application (usually once per frame).

C.2.1 Error Handling

Attribute	Description
`int error;`	The error's ID.
`int process;`	Process ID of game-bot that experienced the error.
`address_t instruction;`	The index of instruction that caused the error.
`address_t stack;`	The stack address that experienced the error.
`char description[128];`	Character string that holds a description of the error.

Table C.3: Public attributes of the 'zbl_error_t' type.

The virtual machine interface of the ZBL-API also includes the definition of the record structure 'zbl_error_t' (see Table C.3) that can be used for reporting run-time errors that occur in game-bot processes of the virtual machine. In addition to this data structure, the ZBL-API also includes a set of functions for error handling that can be used with this data structure.

`int zbl_getErrors(void);`
This method returns the number of unqueried run-time errors that have been recorded in the virtual machine's error list.

`void zbl_resetOnError(int pID);`
This method toggles a flag in the game-bot process with the given process ID that will trigger the process's reset on the occurrence of a run-time error (see also method 'zbl_resetProcess').

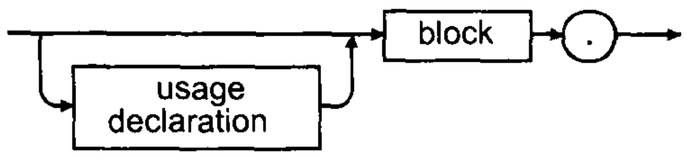
`zbl_error_t zbl_nextError(void);`
`zbl_error_t zbl_peekError(void);`
These methods allow querying of run-time errors in the virtual machine. For this they retrieve the next error from the virtual machine's error list and return them

as a structure of the 'zbl_error_t' record type. While 'zbl_nextError' then removes the error from the virtual machine's error list, 'zbl_peekError' leaves the error list untouched.

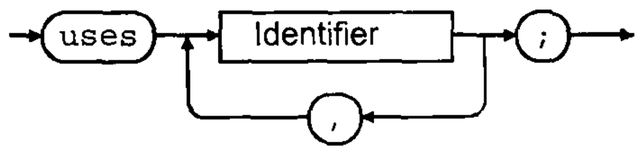
C.3 ZBL/0 Syntax

C.3.1 Core Functionality

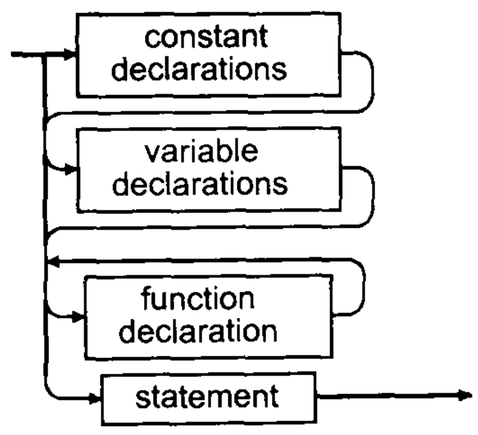
program:



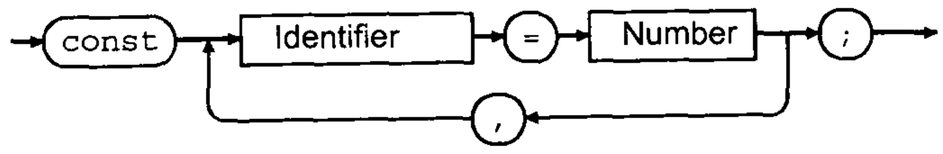
usage-declaration:



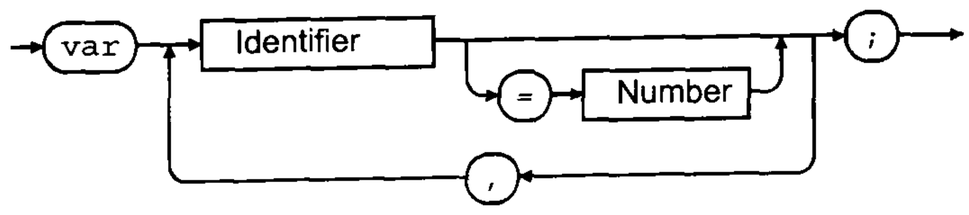
block:



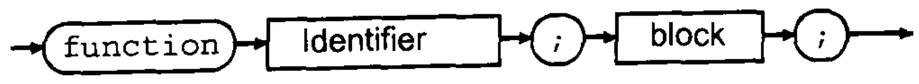
constant-declarations:



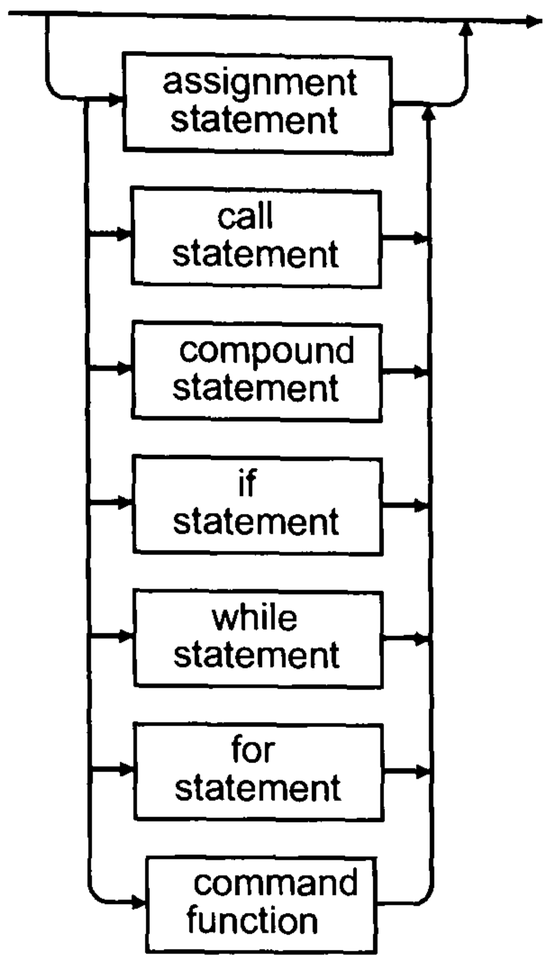
variable-declarations:



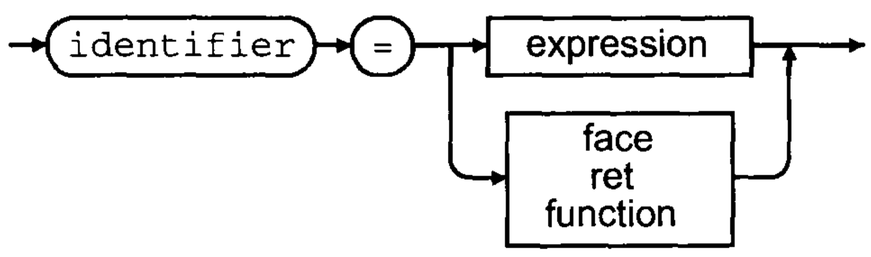
function-declaration:



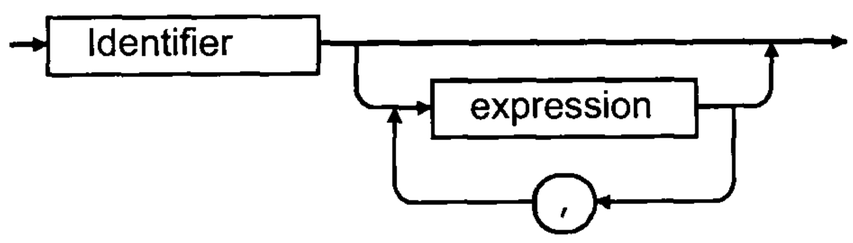
statement:



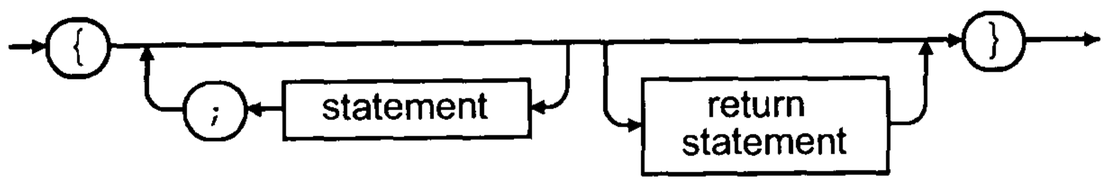
assignment-statement:



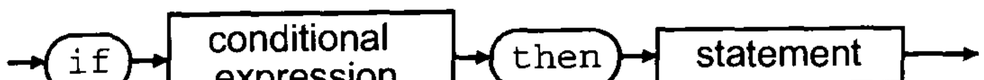
call-statement:



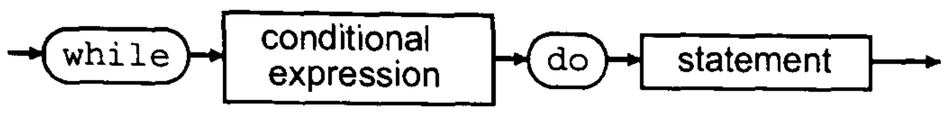
compound-statement:



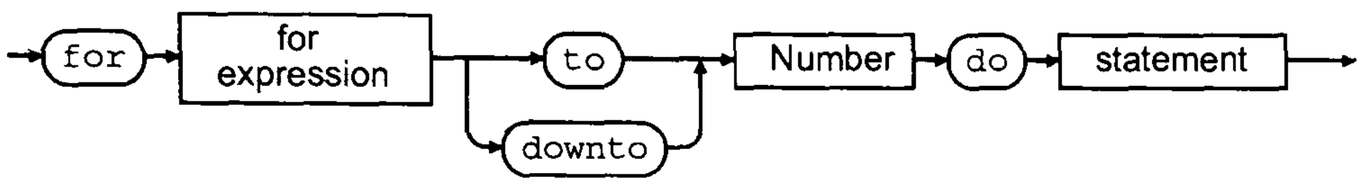
if-statement:



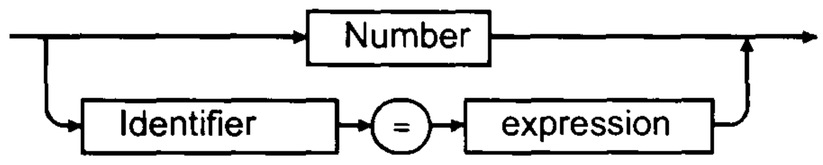
while-statement:



for-statement:

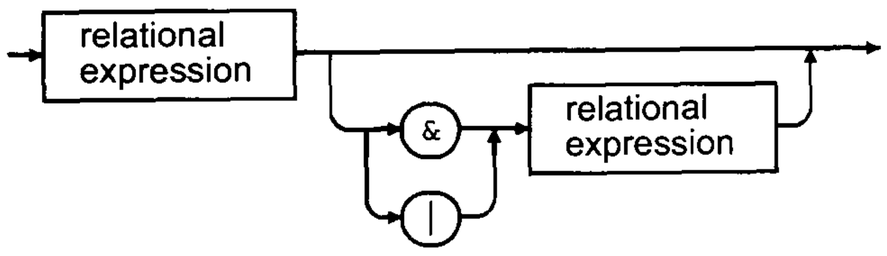


for-expression:

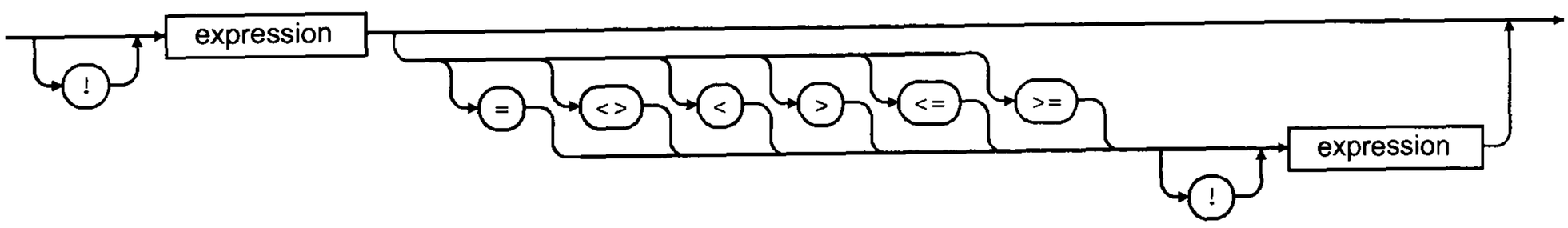


conditional-expression:

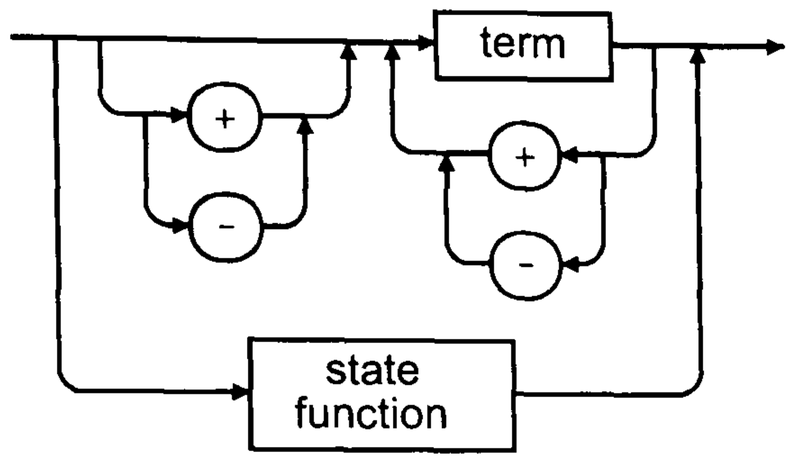
boolean-expression:



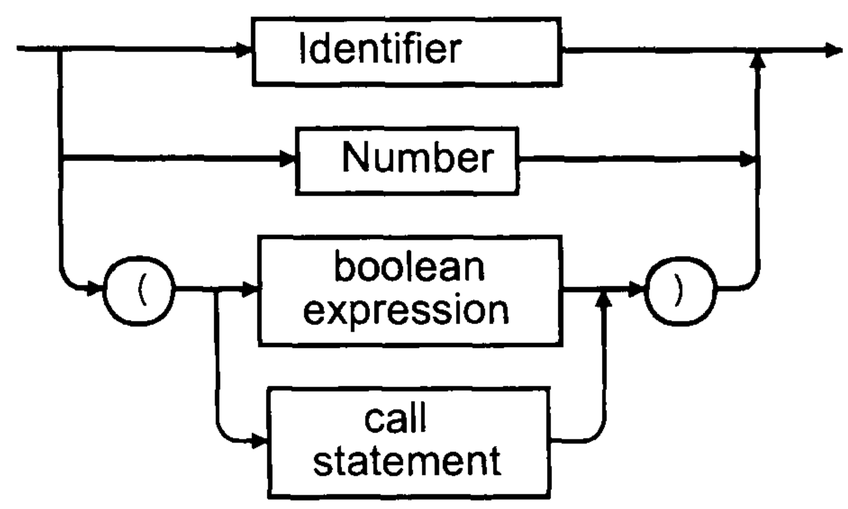
relational-expression:



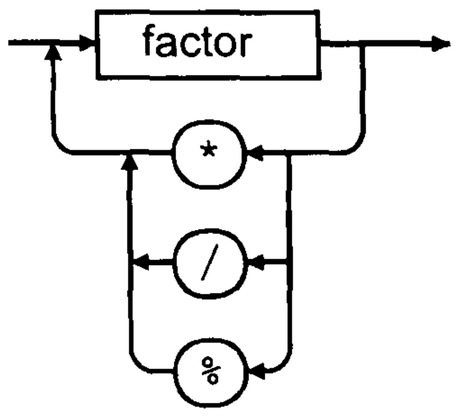
expression:



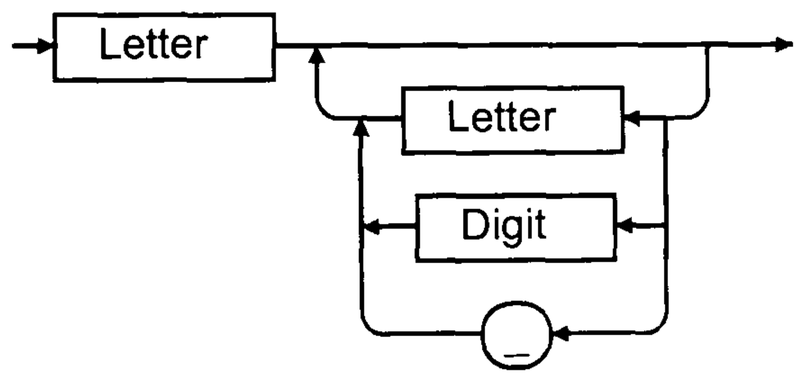
top-level:



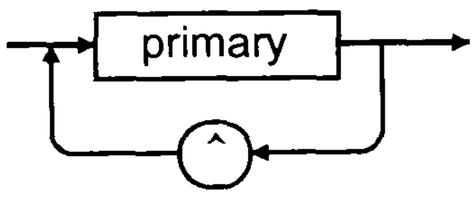
term:



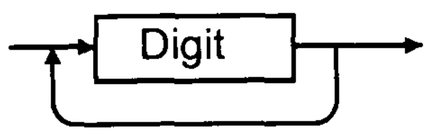
Identifier:



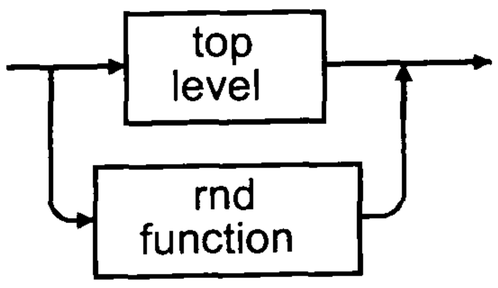
factor:



Number:



primary:

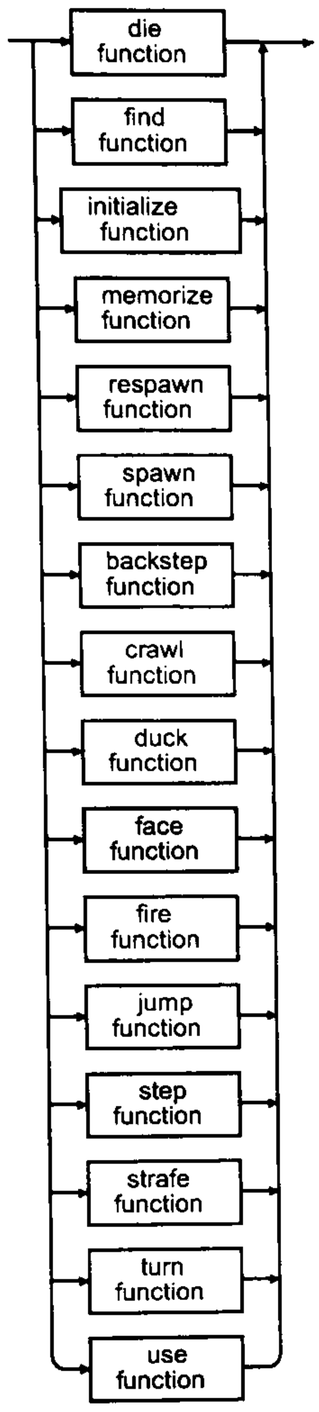


Letter any of the 26 letters of the alphabet (capital or lower case)

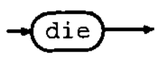
Digit any digit from '0' to '9'

C.3.2 Intrinsic Functions

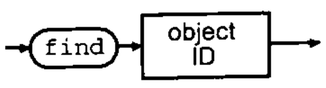
command-function:



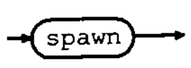
die-function:



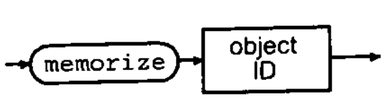
find-function:



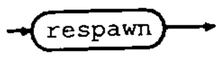
spawn-function:



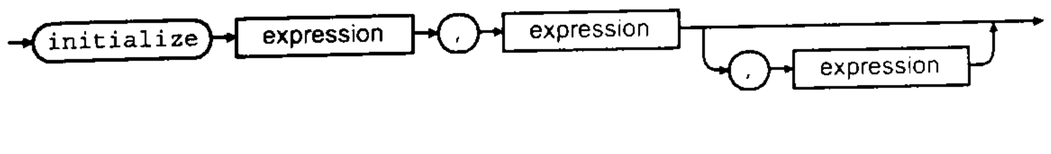
memorize-function:



respawn-function:

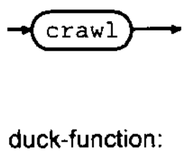


initialize-function:



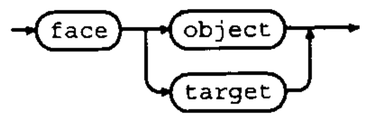
backstep-function:

crawl-function:



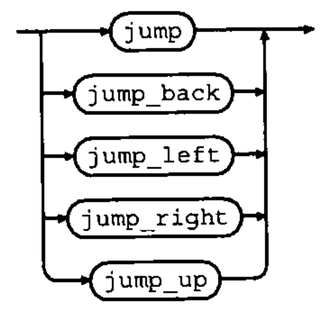
duck-function:

face-function:



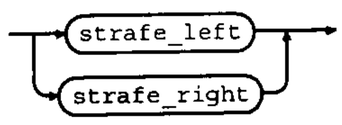
fire-function:

jump-function:

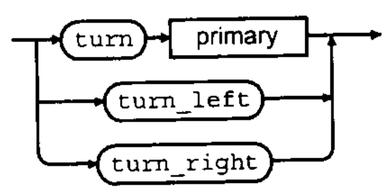


step-function:

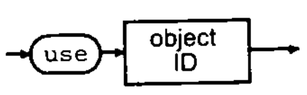
strafe-function:



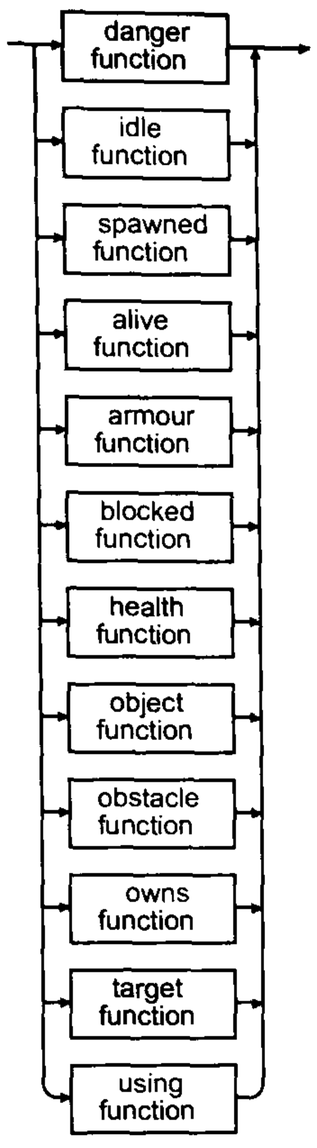
turn-function:



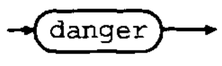
use-function:



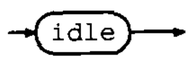
state-function:



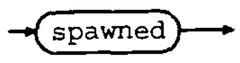
danger-function:



idle-function:

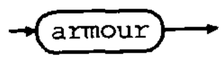


spawned-function:

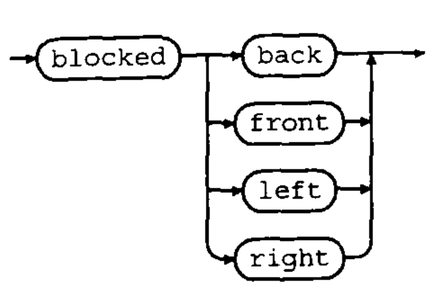


alive-function:

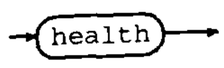
armour-function:



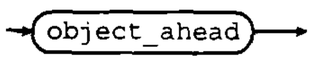
blocked-function:



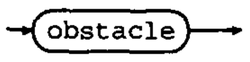
health-function:



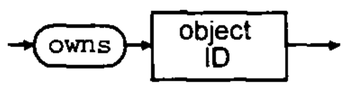
object-function:



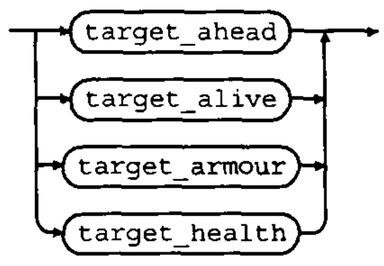
obstacle-function:



owns-function:



target-function:



using-function:

rnd-function:

object-ID:

Appendix D

The AvDL Scripting Language

In Chapter 8 of this thesis we introduced the Avatar Description Language (AvDL) which we developed in our investigation of behaviour definition languages. AvDL is a scripting language that provides mechanisms for behaviour definition for virtual entities in computer games. This appendix presents a detailed description and reference of the syntax and functions of AvDL.

```
import ...  // loading of extensions

entity <name>
{
   ...  // declaration of types and variables

   ...  // definition of functions and methods

   <name>() // program entry function
   {
      ...
   };
};
```

Table D.1: Basic structure of an AvDL program.

D.1 Programming in AvDL

Every AvDL entity program must declare itself as an '**entity**' object which encapsulates the program (in a similar manner to C++ namespaces). The entry point of an AvDL program from which it will begin program execution is the entity object's main method, which is always the final function defined within an entity object (unlike other functions it is terminated with a semicolon). The entity's main method is always given the same identifier as the entity object itself (see Table D.1).

D.1.1 Core Functionality

`action`	`bool`	`break`	`byte`	`case`	`class`	`const`
`continue`	`default`	`delete`	`do`	`double`	`else`	`elsif`
`entity`	`event`	`exit`	`extends`	`finite`	`float`	`for`
`foreach`	`forever`	`from`	`fuzzy`	`getstate`	`global`	`goal`
`if`	`import`	`int`	`long`	`new`	`of`	`onentry`
`onexit`	`plan`	`public`	`reached`	`repeat`	`return`	`scalar`
`select`	`setstate`	`short`	`state`	`switch`	`to`	`trigger`
`triggered`	`typedef`	`unsigned`	`until`	`uses`	`while`	`void`
`volatile`						

Table D.2: AvDL reserved words.

The command syntax of AvDL is based on the C [Kerninghan and Ritchie 1988] and C++ [Stroustrup 1997] programming languages (see Table D.2). All statements in AvDL need to be terminated with a semicolon (';').

LOADING OF EXTENSIONS:

Extension libraries are loaded at the start of the program above the entity's definition using the '**import**' keyword (or its '**uses**' synonym), followed by one or more identifiers (separated by commas) that must match the name(s) of the extension(s).

```
import <identifier> {, <identifier>} ;
```

COMMENTS:

Comments can be considered a basic component of modern programming languages. The AvDL scripting language allows the use of line-comments, as well as block-comments. Line-comments are marked with the '//' (double forward slash) character sequence that is also used for line comments in the C++ programming language. Any character following the line-comment character sequence until the next new line will be ignored.

Example:

```
// this is a line comment
```

or

```
scalar x=1; // and so is this
```

Block-comments are marked with the two character sequences '/*' to open a comment and '*/' to close a comment. The text encapsulated between these two symbols will be ignored by the compiler. These block-comments can span across more than a single line of source code, i.e. if a comment is opened in one line of code and closed in another line of code, all text in between will be commented out.

Example:

```
/* a simple block comment */
```

or

```
/* a block comment spanning
   across three lines of
   source code */
```

D.1.2 AvDL Data Types

All variable data types in AvDL are auto-initialising, i.e. unless variables are initialised explicitly when they are declared, variables will be given a default value. For numerical data types that default value is '0' (zero), while other types will automatically be initialised to the empty value 'NULL'.

Variables that use AvDL's types can be declared (and defined) at the top

of a block or globally using the desired type, followed by one or more variable identifiers (separated by commas).

AvDL allows users to declare their own data type, effectively an alias type, using the '**typedef**' statement.

Example:

```
typedef scalar real; // make "real" an alias for "scalar"
```

Scope in AvDL is handled similar to C/C++: constants and variables that are defined globally (outside of a class or function) can be accessed from anywhere in the AvDL program, whereas constants and variables that are defined locally within a block can only be accessed from inside that block.

TYPELESS DATA:

AvDL provides a typeless data type '**void**' which is identical to the typeless '**void**' found in C/C++. It is an "empty" (non numeric) data type which requires no storage. The main use of this type is as the return type for functions that do not return any values (procedures), as well as the definition of empty parameter lists for functions.

D.1.2.1 Atomic Data Types

Variables of the atomic data types can be supplied with additional information in the form of type qualifiers that determine the access mode of the variables.

If the '**const**' type qualifier is used when a variable is declared, that variable must be initialised, after which its value can no longer be altered, effectively making this variable a constant value.

Example:

```
const scalar five=5; // create a constant named "five"
```

The use of the '**global**' type qualifier for the definition of a function return

type marks the function as "exported" for use by other entities. This is the means by which an entity's functions are advertised, making it an annotated entity.

Example:

```
global void exported(void) // advertise "exported"
{
   statement(); // execute "statement"
};
```

The '**triggered**' (-'**of**') type qualifier is used for binding scalar and Boolean variables to events. AvDL variables that have been declared using the '**triggered**' type qualifier for a given event will be set to the value '1' or '**true**' when that event occurs.

Example:

```
triggered ev of scalar trg; // "trg" is bound to event "ev"
```

The '**volatile**' type qualifier is used in the C and C++ programming languages to mark data that is influenced by processes which are external to the current program. In AvDL the '**volatile**' variable attribute is used to bind data in the host application to scalar and Boolean AvDL variables in entity programs. Variables that are '**volatile**' in AvDL can be mapped to variables in the host application.

Example:

```
volatile scalar var; // "var" is externally accessible
```

NUMERICAL VALUES:

AvDL has a single numeric data type that can take floating point values as well as integer values, which is the '**scalar**' data type. A number of aliases for the '**scalar**' type allow the use of the more commonly known ordinal and floating point data types found in C/C++.

Example:

```
scalar var;        // create variable "var" (initialised to 0)
scalar o=5,r=0.5; // two variables, one ordinal and one real
int    ordinal=5; // "int" alias for "scalar"
float  real=0.5;  // "float" alias for "scalar"
```

BOOLEAN VALUES:

AvDL has a data type for Boolean values. Variables of the '**bool**' data type can take the values '**true**' (mapped to the value '1') or '**false**' (mapped to the value '0'). Boolean values are by default auto-initialised to the value '**false**'.

REFERENCES:

In AvDL there are no pointers to variables. Instead one can create references – like references in C++ – to address a variable. A reference to a variable is similar to having a second identifier for the same variable. References are declared by preceding the variable identifier with the '&' (ampersand) symbol. A reference must be initialised during its declaration (if it is not used as a function parameter) and cannot be changed (redirected to a different variable) during the lifetime of its identifier.

Example:

```
scalar var = 1;
scalar &ref = var; // "ref" is a reference to "var"
```

FUNCTION BINDINGS:

Functions that are defined within the host application can be mapped to the AvDL '**action**' data type. If an '**action**' is declared with an identifier only, the AvDL virtual machine will expect the same identifier to be used for the function in the host application which is mapped to the AvDL action. If a name is explicitly provided during the action declaration, that "name" is expected to be the identifier used for the function in the host application. If the function in the host application expects parameters, the action can be declared using an optional parameter list.

```
action <identifier>[(<parameters>)];
```

or

```
action <identifier>(<name>[,<parameters>]);
```

Example:

```
action func; // bind function "func"
```

D.1.2.2 Data Structures

Apart from the basic data types that are part of the AvDL scripting language, AvDL allows users to use more complex data structures, some of which are derived from the atomic data types.

ARRAYS:

Arrays are the simplest form of aggregate data type, able to hold a number of data elements of the type that they are derived from (all array elements have the same data type). In the current AvDL specification, all types of arrays are restricted to a single dimension. There are three types of arrays in AvDL: static arrays, dynamic arrays and associative arrays.

The index value for the first field in static and dynamic AvDL arrays is '0'. Fields in associative arrays are not usually addressed using numbers. but use named indices (associations) instead.

Static arrays in AvDL are defined in a similar manner to arrays in C. These arrays are variables of any AvDL data type that have been declared using the subscript operator ('[]') with a size (of array) indicator as their suffix. The elements stored in an array's fields are by default auto-initialised to the value '0'.

Example:

```
scalar array[5]; // a static array with 5 fields
```

Dynamic arrays in AvDL are variables of any AvDL data type that have been declared using the subscript operator ('[]') without a size indicator and which are then given a size using the '**new**' operator (see section C.1.3). Dynamic arrays are by default auto-initialised to the empty value 'NULL' when they are first created.

Example:

```
scalar dynamicArray[]; // dynamic array, pre-initialised to "NULL"
       dynamicArray = new scalar[5]; // allocate 5 fields
```

The memory that has been allocated for dynamic arrays using the '**new**' operator must be freed again, using AvDL's '**delete**' operator. Unlike the '**delete**' operator in C++, the AvDL operator does not require the use of the subscript operator to free dynamically allocated arrays.

Example:

```
delete dynamicArray; // delete and reset to "NULL"
```

Associative arrays in AvDL are variables of any AvDL data type that have been declared using the subscript operator ('[]'), i.e. their declaration is identical to that of regular dynamic arrays in AvDL. Associations are created dynamically as soon as they are used in the source code for the first time. Externally associations are identifiers that are used as indices for array fields. Internally each new association is given an increasing index value. Associative arrays are by default auto-initialised to the empty value 'NULL' when they are first created. Fields in associative arrays that are created but not assigned an element to hold are by default auto-initialised to the value '0'.

Example:

```
scalar associativeArray[]; // pre-initialised to "NULL"
       associativeArray[first] = 5; // first field set to 5
       associativeArray[second];    // second field created
       associativeArray[second] = 3; // second field set to 3
```

CLASS STRUCTURE:

In AvDL data structures of the type '**class**' are used for the definition of objects (in the sense of object orientation) as well as records. A class definition defines a new data type, instances of which can be used as variables.

Class structures have two different types of members, attributes and methods. Attributes are member variables of a class, whereas methods are member

functions of the class. Like classes in C++, AvDL classes allow the definition of a class constructor and a class destructor, both of which are special methods that are invoked respectively when an object is instantiated or destroyed.

There are no inline functions, i.e. although methods are declared within the class definition they have to be defined below the class definition itself.

```
class <identifier>
{
   <type> <attribute>; // declare a data member
   <type> <method>(<parameters>); // function member
   ...
   <class ID>();  // constructor declaration
   ~<class ID>(); // destructor declaration
};
```

All classes have one implicit attribute, which is a reference to the current instance of itself, which can be accessed through the use of the 'this' object. The 'this' object is also a hidden parameter which is implicitly passed to all methods of the class.

Classes are also AvDL's implementation of the **record** aggregate data structure, if they only define attributes but no methods. In that case classes are used to group variables consisting of a combination of types together in a single entity which can then be referred to through a single identifier, i.e. the class's name.

Finally, AvDL also allows the use of an implicit class, i.e. a class definition that is stored within an external file. The file containing an implicit class definition must either be an AvDL source code file containing only the class definition or alternatively a pre-compiled class definition as bytecode for the virtual machine. Once loaded the implicit class can be used like any other class.

```
class <identifier> = "<filename>";
```

EVENT TYPE:

The '**event**' data structure is linked to events occurring in the AvDL system's virtual machine. It provides an event handler for the current entity program and

as such requires the definition of an AvDL instruction list (in a block of instructions) that will be executed once the event it has been defined for occurs.
`event <identifier> <block>;`

Example:

```
event X // event handler for event "X"
{
   statement1(); // execute "statement1"
   statement2(); // execute "statement2"
};
```

FINITE STATE MACHINE STRUCTURE:

State machines are a tried and tested AI technology which has been proven suitable for many kinds of computer games. They are the most used AI technology in computer games as they allow for simple definition of deterministic behaviour.

The AvDL '**state**' data structure allows the definition of state machines in AvDL entity programs. To create a finite state machine it should be declared with the '**finite**' state qualifier, but this may be omitted as finite state machines are the default state machine type in AvDL. The declaration of finite state machines in AvDL shares elements with the definition of a '**class**' data structure, as its members are declared within the structure similar to an object's methods and then defined outside of the structure itself.

A finite state structure has up to three specialised function members. These methods are an entry function ('**onentry**'), an exit function ('**onexit**') and the state's body (sharing its identifier with the state structure itself). Of these methods only the state's body must be defined, while the entry and exit functions are optional. The state's body should be provided with a transition target, marking the current state structure's follow state. If no transition target is provided, the value 'NULL' will be used by default, which will terminate the execution of the state machine when the state transition occurs.

Other members of a state structure will always be treated as states by the FSM (independent of their actual types).

Each state in a state structure needs to be provided with a "next" state (transition target) to declare which state the currently active state will change into

when it finishes.

```
[finite] state <identifier>
{
   <type> <identifier>[()], <transition>; // state
   ...
   <onentry>(); // entry function
   <onexit>();  // exit function
   <identifier> () [, <transition>]; // state body
};
```

The finite '**state**' structure is similar to unions in C, as at any one time only one state within it will be fully active. Its members can also can be used as identifiers for states in a similar manner to the named constants of C enumerated data types.

FUZZY STATE MACHINE STRUCTURE:

Structures of the fuzzy '**state**' data type in AvDL are declared with the '**fuzzy**' type qualifier. The fuzzy state structure is similar to a record data structure, as it holds several data members. These members can either be scalar values or references to other fuzzy state structures, each of which can be provided with an optional weight. Like the data members of a finite state structure the members of fuzzy state structures can be used as identifiers for states in a similar manner to the named constants of C enumerated data types.

```
fuzzy state <identifier>
{
   <type> <identifier> [ @ <weight> ]; // state
   ...
};
```

GOAL DATA TYPE:

AvDL provides the data type '**goal**' for goal-oriented behaviour. There are two ways in which this data type can be used. In its simplest form, a goal is defined as a single variable of the '**goal**' type that has been assigned an expression defining the preconditions that have to be satisfied for the goal to be reached.

```
goal <identifier> = <expression>;
```

Optionally a priority (weight value) for the planner can be set for the goal. If no priority is explicitly set, it will by default be set to the value '1.0'.

```
goal <identifier> @ <priority> = <expression>;
```

Example:

```
goal trueX = (x==true);
```

or

```
goal trueX @ 1.0 = (x==true);
```

The second method uses the '**goal**' type as a compound data structure grouping several data members, each being a separate named precondition for the satisfaction of the goal. These named preconditions are effectively sub-goals of the goal structure.

```
goal <identifier>
{
   <precondition>:  <expression>;
};
```

The '**goal**' structure as well as each of its member preconditions can optionally be supplied with a priority for the planner (by default set to '1.0').

```
goal <identifier> @ <priority>
{
   <precondition> @ <priority>:  <expression>;
   <precondition> @ <priority>:  <expression>;
};
```

Priority	Symbol	Description
1	[]	subscript
	.	member access
2	plan	plan generation
	reached	goal state query
	setstate	state access
	getstate	state query
	trigger	event trigger
3	!	logical negation
	+	positive
	-	begative
	~	length/size
	++	increment
	--	decrement
	new	allocate memory
	delete	free memory
4	/	division
	*	multiplication
	%	modulo
5	+	addition
	-	subtraction
6	==	equality comparison
	!=	non-equality comparison
	<	less-than comparison
	<=	less-or-equal comparison
	>	more-than comarison
	>=	more-or-equal comparison
7	&&	logical AND
	||	logical OR
8	=	value assignment
	/=	compound assignment (division)
	*=	compound assignment (multiplication)
	%=	compound assignment (modulo)
	+=	compound assignment (addition)
	-=	compound assignment (subtraction)
9	,	concatenation

Table D.3: AvDL operator precedence.

D.1.3 Operators

Other than the standard arithmetic and logical operators (see Table D.3), AvDL also provides a number of special operators for use with its game AI data structures.

- **new**
  The operator '**new**' is used for allocating memory for dynamic arrays.

- **delete**
  The operator '**delete**' is used for freeing allocated memory from dynamic arrays.

- **setstate**
  For FSMs the '**setstate**' operator is used to set any state or state member to be the currently active state, triggering a state transition. In FuSMs it is used to set state members, optionally allowing the specification of a weight value for the state member.

- **getstate**
  The operator '**getstate**' returns a reference to the currently active FSM state, effectively allowing the currently set state to be queried.

- **trigger**
  The operator '**trigger**' is used for spawning events from within entity programs. Events can optionally be addressed directly towards a specific entity.

- **reached**
  The operator '**reached**' is used for testing a goal for completion, i.e. for testing the goal state.

- **plan**
  The operator '**plan**' directly operates on goals, generating a plan from all valid goals in an entity program.

D.1.4 Control Structures

Blocks containing sequences of statements in AvDL are similar to blocks in the C/C++ programming languages. They can be created by inserting the statement

sequence between opening braces ('{') and closing braces ('}').

Example:

```
{
    statement1();
    statement2();
    statement3();
}
```

D.1.4.1 Selections

AvDL contains all of the conditional statements found in the C/C++ programming languages, as well as a number of additional selection methods. Within any expression in a selection, a value of '0' (zero) will be interpreted as '**false**' (condition not satisfied), while any other value will be interpreted as '**true**' (condition satisfied).

SIMPLE SELECTIONS:

The **if statement** is used to determine whether or not a particular function, expression or control structure is to be executed. It can be used to express monadic (one-alternative) or dyadic selections. In the latter case an **else clause** can be used in combination with the '**if**' to provide two alternatives.

```
if(<expression>) <block/statement;>
```

or

```
if(<expression>)
    <block/statement;>
else
    <block/statement;>
```

Example:

```
if(b==true) statement1();
```

or

```
if(b==true)
    statement1();
else
```

```
statement2();
```

MULTIPLE SELECTIONS:

AvDL includes a more complex if statement in the form of the **elsif statement** which provides a further condition, allowing the expression of selections with up to three alternatives (if used with '**else**').

```
if(<expression>)
   <block/statement;>
elsif(<expression>)
   <block/statement;>
```

or

```
if(<expression>)
   <block/statement;>
elsif(<expression>)
   <block/statement;>
else
   <block/statement;>
```

Example:

```
if(b==true)
   statement1();
elsif(b==false)
   statement2();
```

or

```
if(x<5)
   statement1();
elsif(x>5)
   statement2();
else
   statement3();
```

For multiple condition switching the **switch statement** with **case clauses** is used. A '**default**' clause can be added to address conditions which are not

explicitly covered by a '**case**'.

The values marking a '**case**' must be constant values, i.e. they must not be variables. If a value and the selection expression in the '**switch**' statement match, all instructions following that particular case clause will be executed until the end of the structure is reached, i.e. there is a fall-through after every case. If no values match any of the given cases then any instruction following the '**default**' keyword will be executed. If no values match the given cases and there is no default given inside the switch structure, the whole switch structure will be ignored and program execution will resume below the structure.

```
switch(<expression>)
{
   case <constant>:  <statement>;
   ...
   [default:  <statement>;]
}
```

An alternative multiple condition that does not have fall-throughs is the **select statement** if it is used with **case clauses**. This '**select**' statement is identical to the switch statement in all aspects except that instructions following a case clause will be executed only until the start of the next case clause, i.e. there is no fall-through after every case.

```
select(<expression>)
{
   case <constant>:  <statement>;
   ...
   [default:  <statement>;]
}
```

AvDL includes a second type of '**select**' statement that allows the multiple selection of ranges of values to identify the statement that is to be executed.

```
select <variable> from
{
   <constant> [to <constant>]:  <statement>;
   ...
```

```
}
```

D.1.4.2 Iterations

AvDL provides several alternative loop statements, several of which are head-controlled loops, while others are foot-controlled. Within any expression in a loop, a value of '0' (zero) will be interpreted as '**false**', while any other value will be interpreted as '**true**'.

HEAD-CONTROLLED LOOPS:

The simplest form of iteration in AvDL programs is the basic **while loop** as found in C. It consists of the '**while**' keyword followed by a conditional expression and a block or a single statement.

```
while(<expression>) <block/statement;>
```

Example:

```
// a simple counter
i=0;
while(i<10)
{
   i=i+1;
}
```

or

```
i=0;
while(i<10)
   i++;
```

A bounded iteration – effectively a more complex version of the while loop – for use in operations that require a known number of iterative steps is the **for loop**. This loop is controlled by three expressions, the second of which is the termination condition of the loop, controlling the iteration. The first and third expressions are optional, providing a means for loop initialisation and adjustment respectively.

```
for(<expr1> ; <expr2> ; <expr3>) <block/statement;>
```

Example:

```
// a simple counter
for(i=0; i<10; i++)
    ;
```

The final head-controlled loop is the **foreach loop** for use with arrays. This loop iterates through all fields of an array, which is especially useful for associative arrays that may have an unknown number of fields.

```
foreach(<variable> of <array>) <block/statement;>
```

Example:

```
scalar aArray[];// associative aray
scalar i; // counter variable
...
foreach(i of aArray)
{ // iterate through "aArray"
   aArray[i]=1;
}
```

FOOT-CONTROLLED LOOPS:

The repetitive **do-while loop** in AvDL is identical to the foot-controlled loop in C/C++. This loop checks its exit condition at the end of the structure. The contained instructions are executed at least once and a new cycle is only entered if the exit condition evaluates as true. Only if the exit condition evaluates as false will the loop be exited. The exit condition must be terminated (using the terminator symbol ';').

```
do <block/statement;> while(<expression>);
```

Example:

```
// a simple foot-controlled counter
i=0;
do
{
    i=i+1;
```

```
} while(i<10);
```

AvDL provides a second repetitive loop, the foot-controlled **repeat-until loop**. This loop also checks its exit condition at the end of the structure. The contained instructions are executed at least once and a new cycle is only entered if the exit condition evaluates as false. Only if the exit condition evaluates as true will the loop be exited, i.e. it repeats until the exit condition is met. The exit condition must be terminated (using the terminator symbol ';').

```
repeat <block/statement;> until(<expression>);
```

Example:

```
// a simple foot-controlled counter
i=0;
repeat
{
   i=i+1;
} until(i==10);
```

AvDL also includes a continuous (never ending) loop, the **do-forever loop**. This has an entrance but no exit and thus does not check for any exit conditions. It keeps cycling through all the instructions contained within until the program is terminated.

```
do <block/statement;> forever;
```

Example:

```
do
{
   statement();
} forever;
```

The use of this iterative structure is equivalent to the use of a while loop whose conditional expression always evaluates as true.

Example:

```
while(1)
{
   statement();
}
```

D.1.5 Commands & Functions

In some instances it may be necessary to jump out of one of the program flow control structures or to ignore some of the statements within a control structure. For this AvDL provides several special statements.

- The '**break**' statement is used if one wants to completely jump out of a loop or a switch/select construct.

  Example:

```
salar i=0;
do // loop for 10 iterations
{
   i=i+1;
   if(i==10) break;
} forever;
```

- The '**continue**' statement is used if one wants jump to the next execution of a loop without executing the following instructions within the loop first. Invoking the '**continue**' statement effectively jumps straight into the next iteration of the loop.

  Example:

```
salar i=0;
while(i<=10) // loop for 10 iterations
{
   i=i+1;
   continue; // next iteration
   i=i-1;    //never executed
}
```

- Programs can be terminated at any time using the '**exit**' statement. Invoking '**exit**' ends program execution and returns the program's exit status to the AvDL virtual machine. This exit status can be given explicitly as an expression. If it is omitted, a successful exit status is returned by default.

```
exit;
```

or

```
exit <expression>;
```

Example:

```
void function(void)
{
   ...
   exit;
}
```

FUNCTIONS & METHODS:

Functions in AvDL programs are defined much like functions in C programs. A function has a name (its identifier) a return data type and (optionally) a list of parameters. The function identifier must not start with a digit. Like C, AvDL allows the forward declaration of functions using prototypes. In function prototypes only the return data type of the function and the data types of parameters need to be stated – identifiers for the parameters only have to be used in the actual function definitions.

It is not possible to declare a function locally, i.e. within the definition of another function.

The forward declaration of the methods of compound data structures is the declaration of the methods within their parent data structure. The definition of the methods usually happens below the definition of the parent data structure but is otherwise nearly identical to the definition of a regular function.

The scope of a function is anywhere below the declaration of the function within the program source code file in which the function has been declared.

Functions do not have to be forward declared, i.e. the declaration and definition of a function can be carried out in a single step.

```
<return type> <identifier>(<parameters>) <block>
```

Example:

```
void function(void)
{
    ...
}
```

The scope of local variables that are declared inside of a function is restricted to that function, i.e. they are known to the AvDL run-time system exclusively within that function. Outside the function they are invalid.

Functions can be jumped out of and values can be returned from within a function to the next higher level using the '**return**' statement.

```
return;
```

or

```
return <expression>;
```

Example:

```
void function(void)
{
    ...
    return;
}
```

Returned values or variables have to be of the same type as the return data type of the function. Variables that receive a return value from a function have to be of the same data type as the function's return data type.

Example:

```
scalar function(void)
{
    ...
    return 1;
}
```

D.1.6 Object Orientation

Object orientation in AvDL is similar but not identical to object orientation in programming languages like C++ and Java. The data structure for object orientation in AvDL is the '**class**' data structure. Unlike classes in C++, classes in AvDL have no mechanism for data hiding, i.e. all methods (member functions) and attributes (data members) of a class are publicly accessible.

D.1.7 AvDL Standard Functions

The current Version of AvDL has a very small number of standard functions which are used in conjunction with the annotation of virtual entities.

scalar **getEntity** (*scalar*)
This function takes the unique ID of an exported function as its parameter and returns the ID of the entity that exported the function.

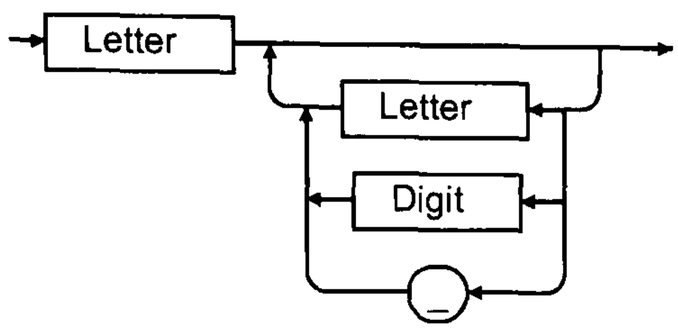
scalar **getGlobal** (*constant string*)
This function takes the name of an exported function as its parameter and returns the ID of a matching exported function if it exists or 'NULL' if it cannot find a match. If called from an event handler, only the entity that caused the event to be spawned will be searched for a matching exported function.

void **setBroadcast** (*void*)
This function asks the run-time environment to advertise an entity's exported functions.

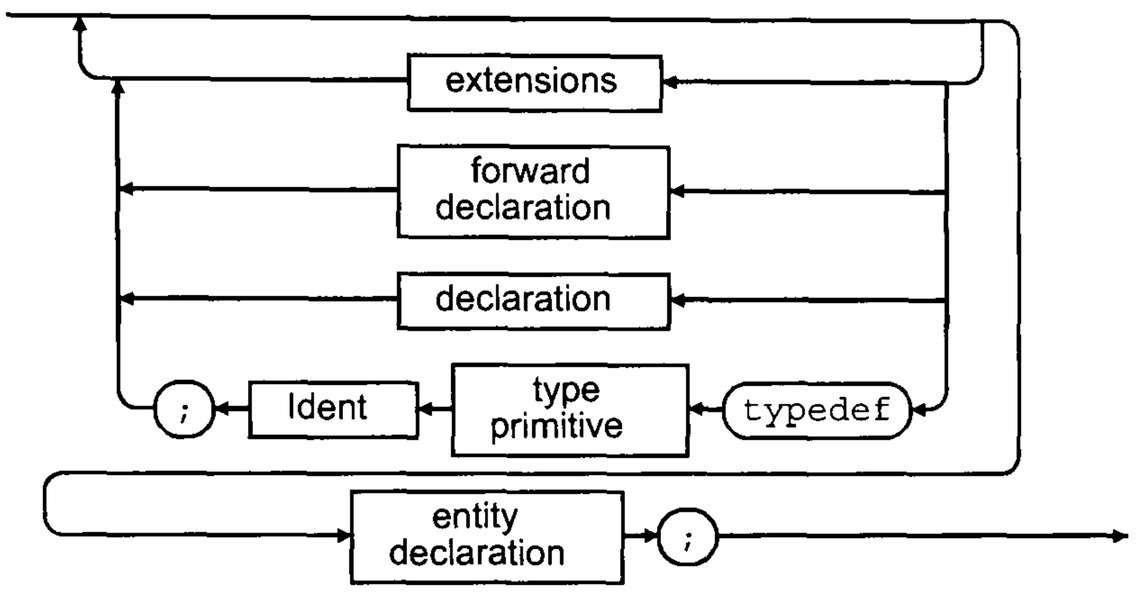
void **setSilent** (*void*)
This function asks the run-time environment to stop advertising an entity's exported functions.

D.2 AvDL Syntax

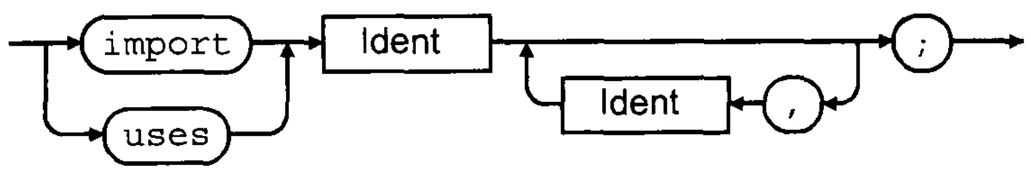
Ident:



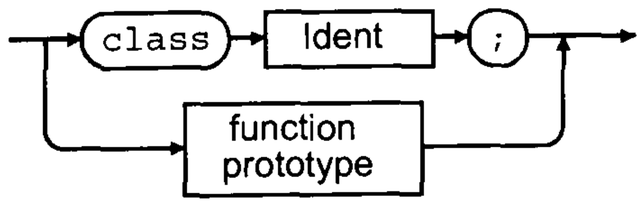
program:



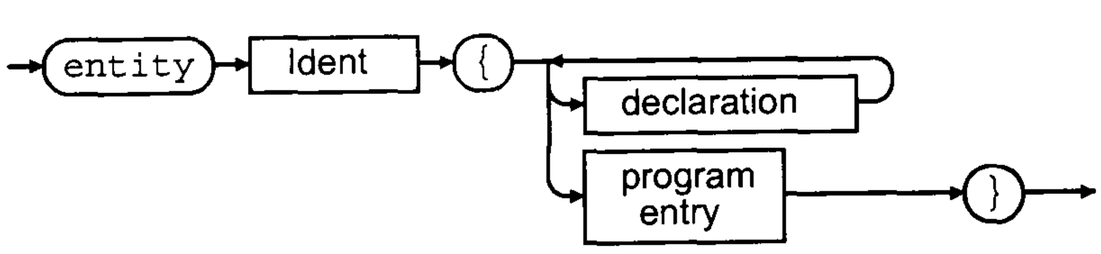
extensions:



forward-declaration:

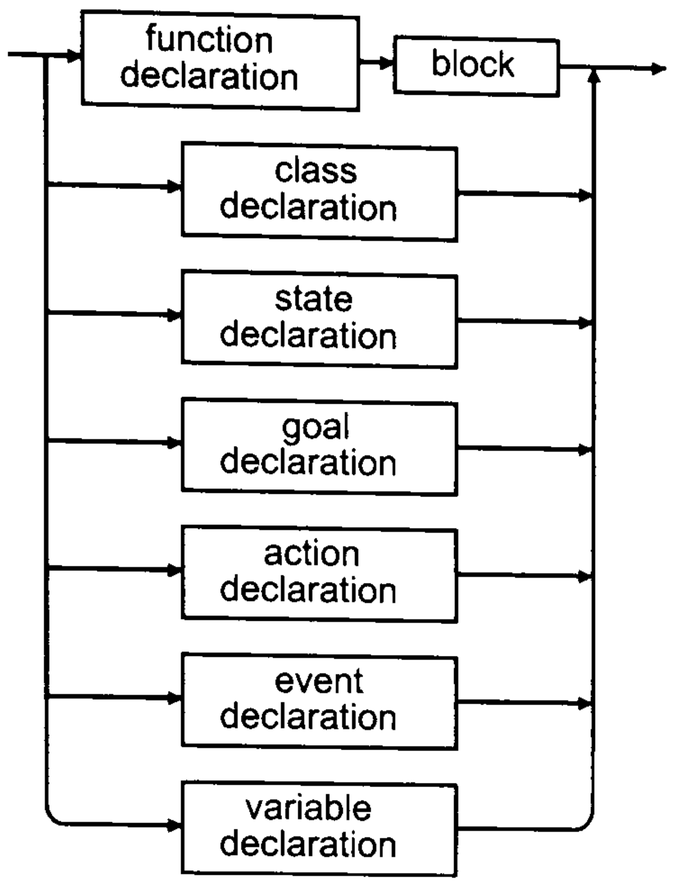


entity-declaration:

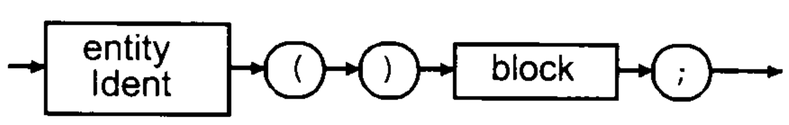


Letter any of the 26 letters of the alphabet (capital or lower case)

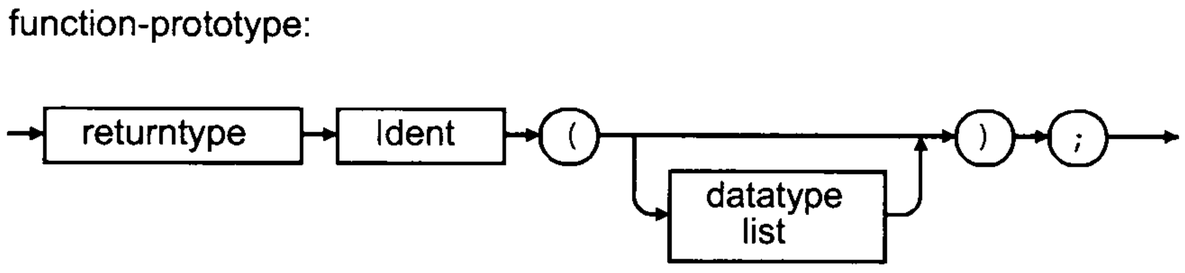
declaration:



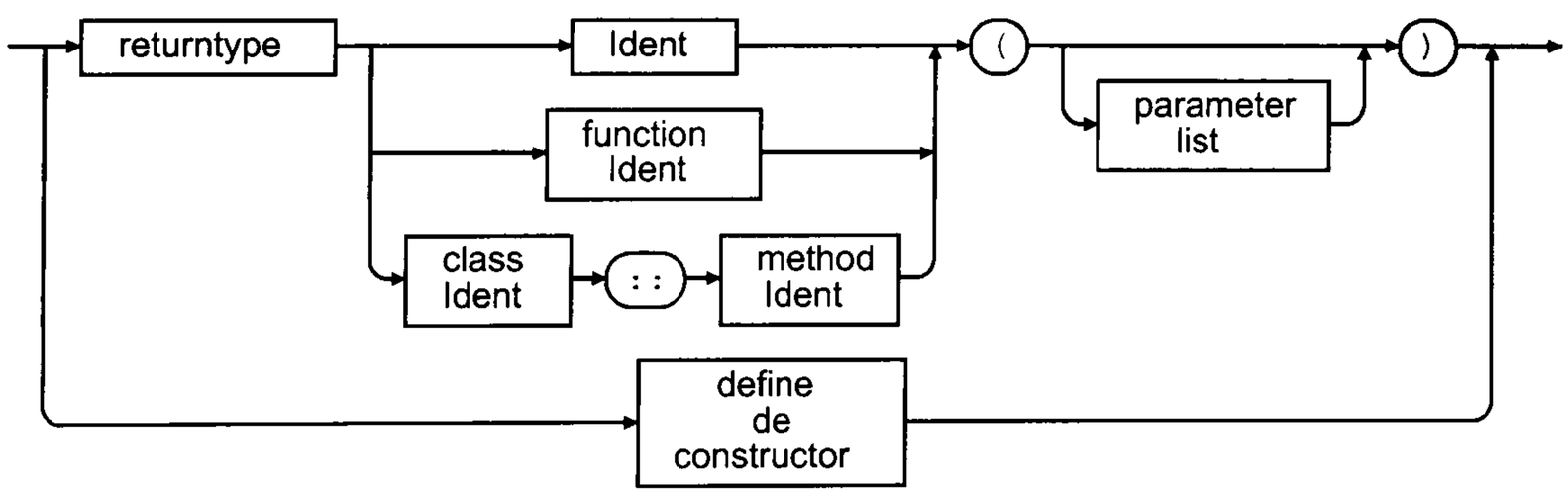
program-entry:



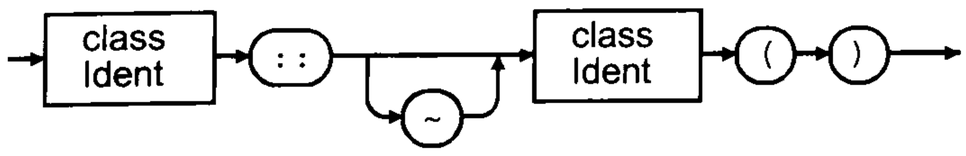
function-prototype:



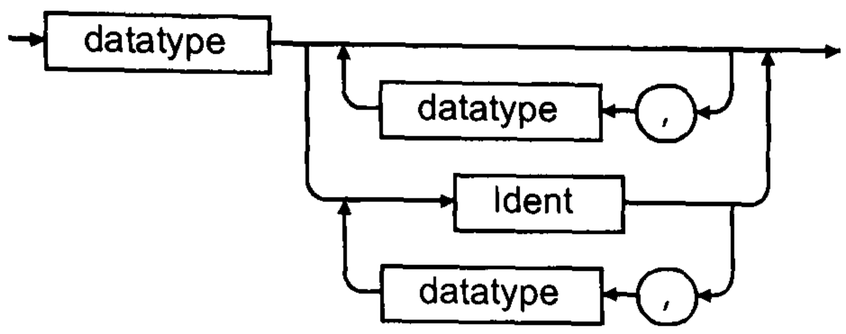
function-declaration:



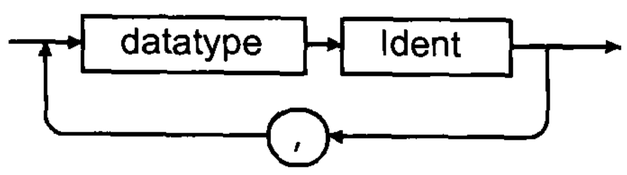
define-de-constructor:



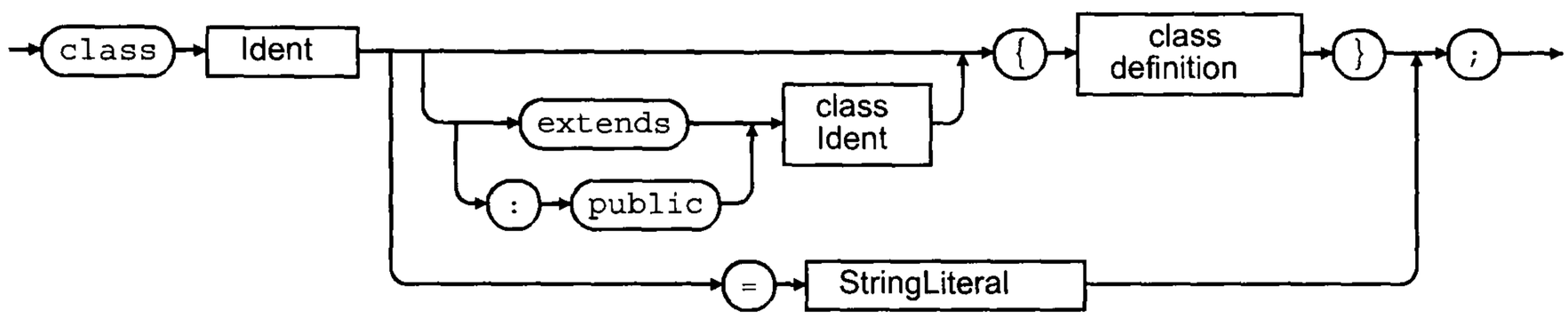
datatype-list:



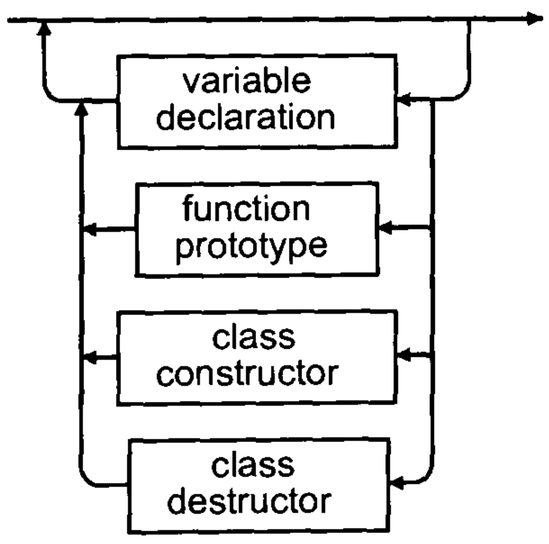
parameter-list:



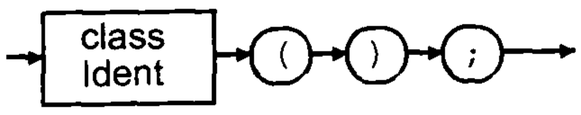
class-declaration:



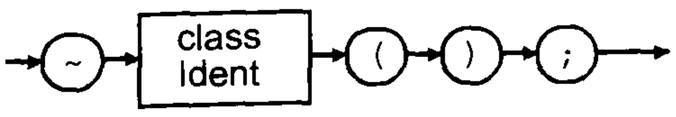
class-definition:



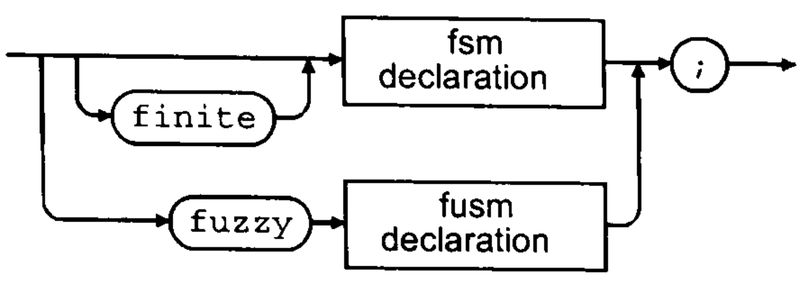
class-constructor:



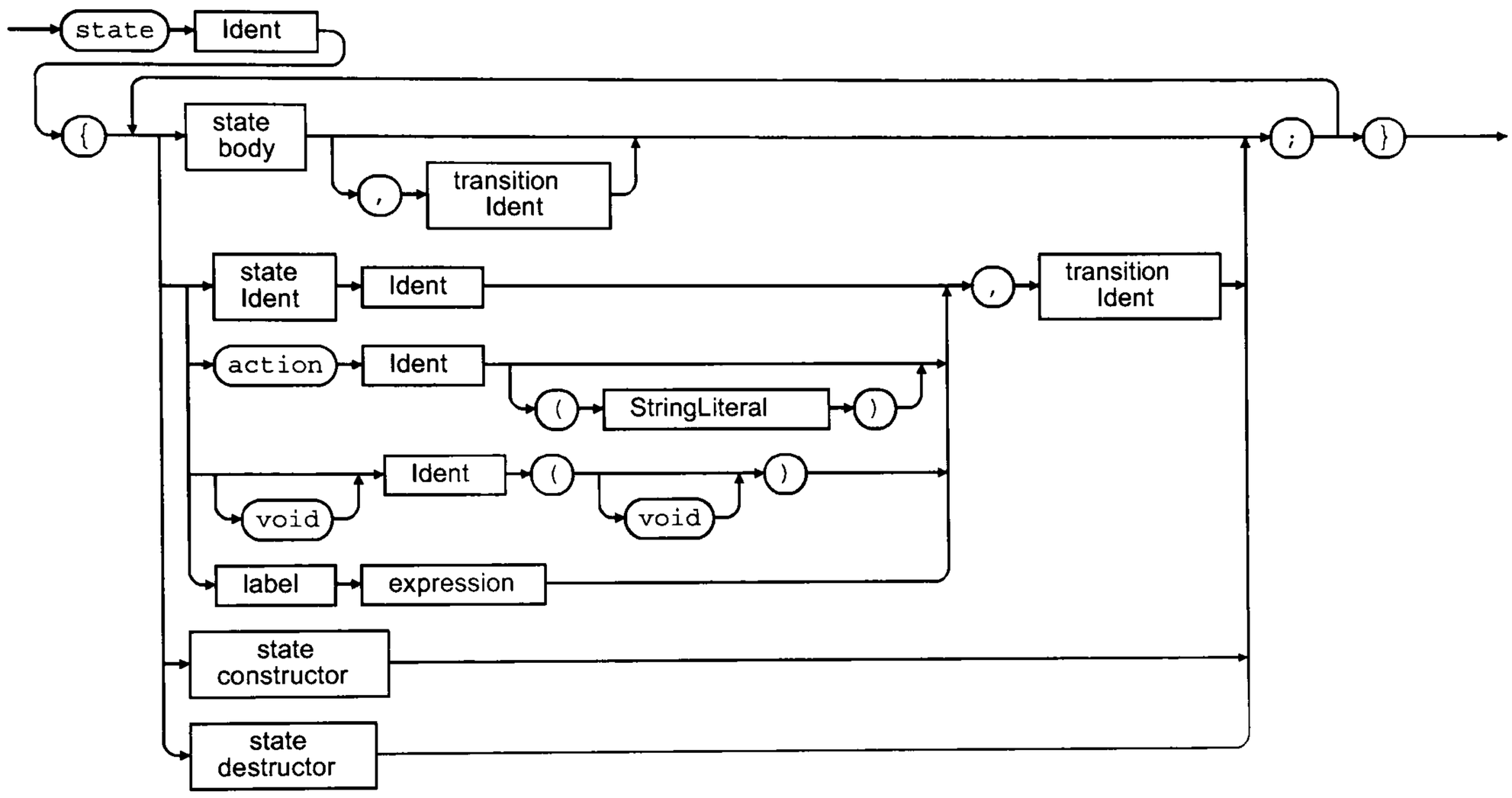
class-destructor:



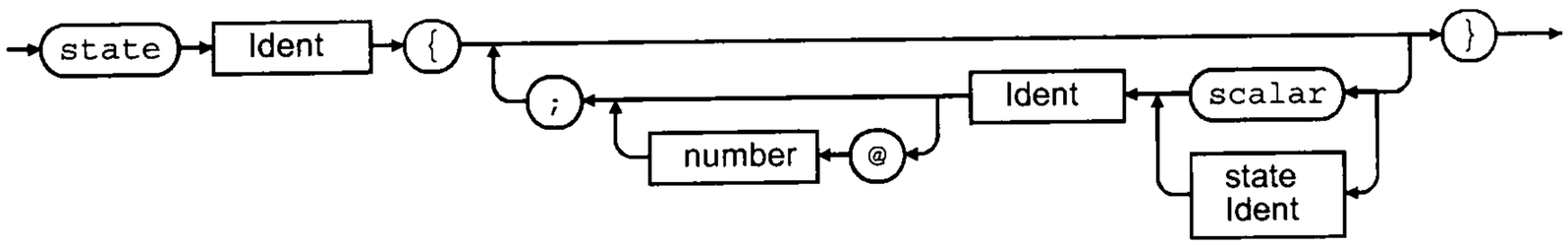
state-declaration:



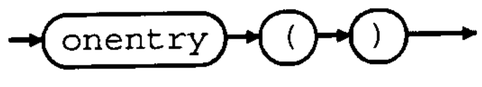
fsm-declaration:



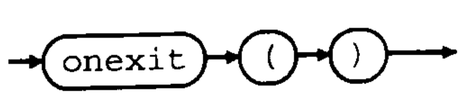
fusm-declaration:



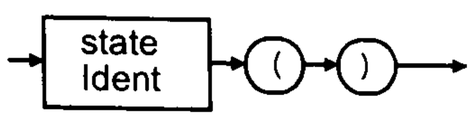
state-constructor:



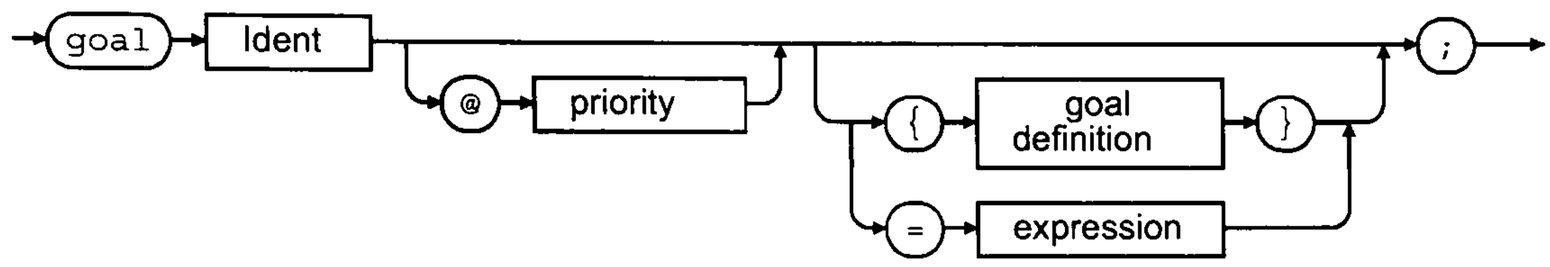
state-destructor:



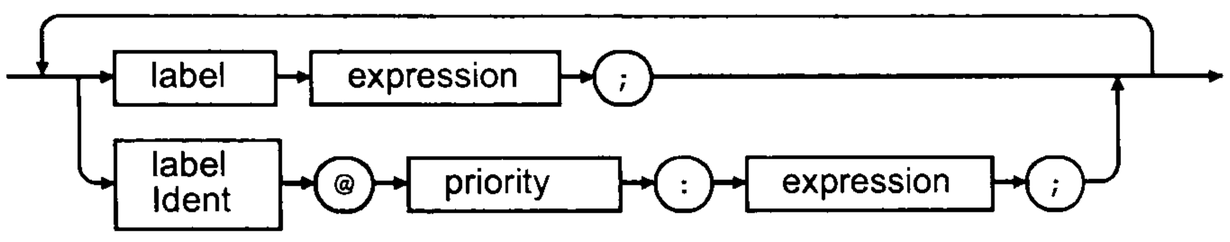
state-body:



goal-declaration:

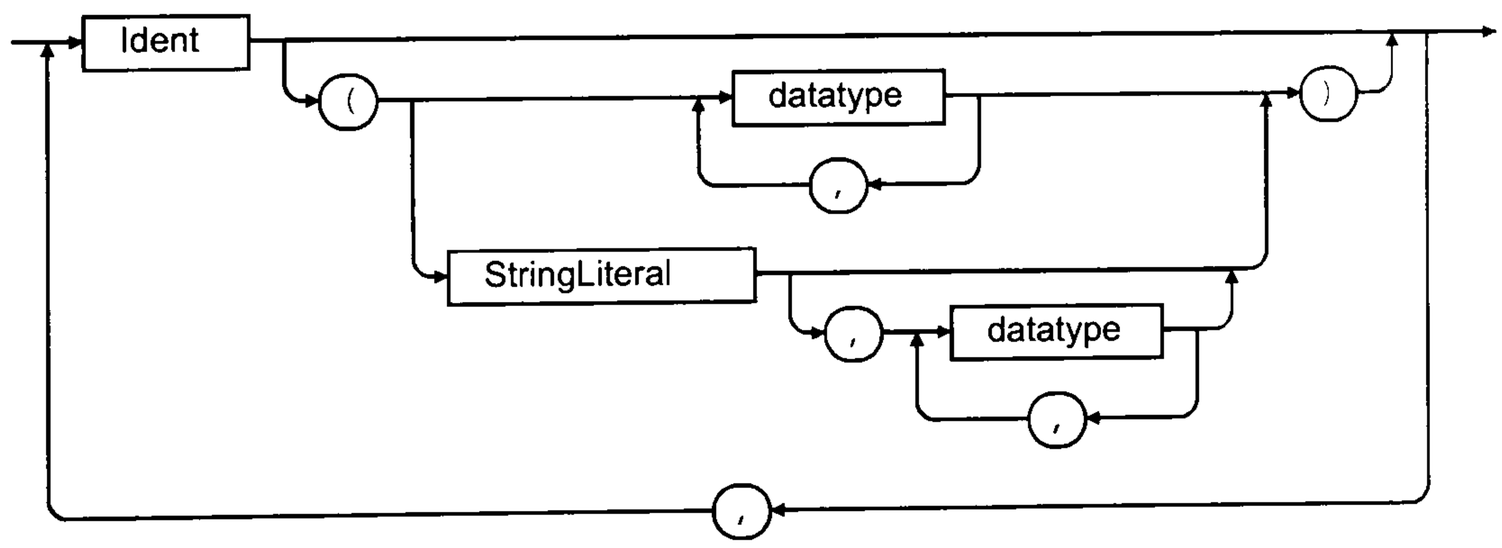


goal-definition:

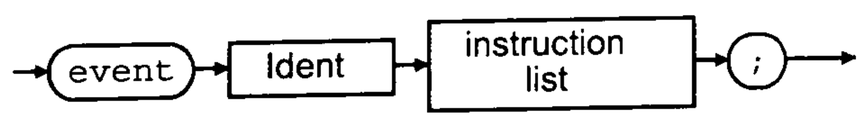


action-declaration:

action-list:



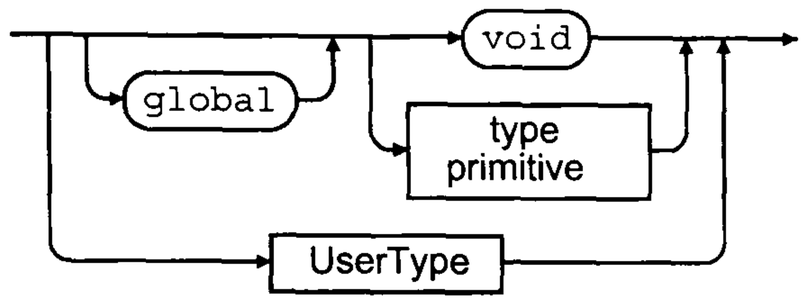
event-declaration:



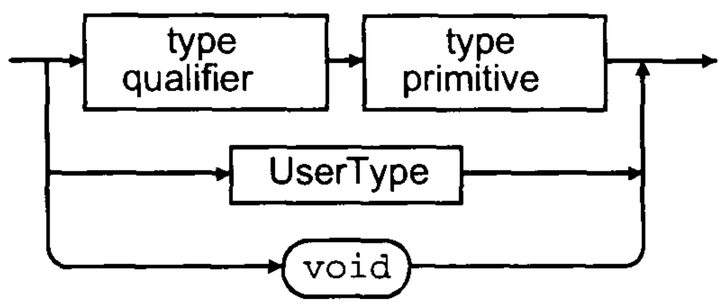
instruction-list:

variable-declaration:

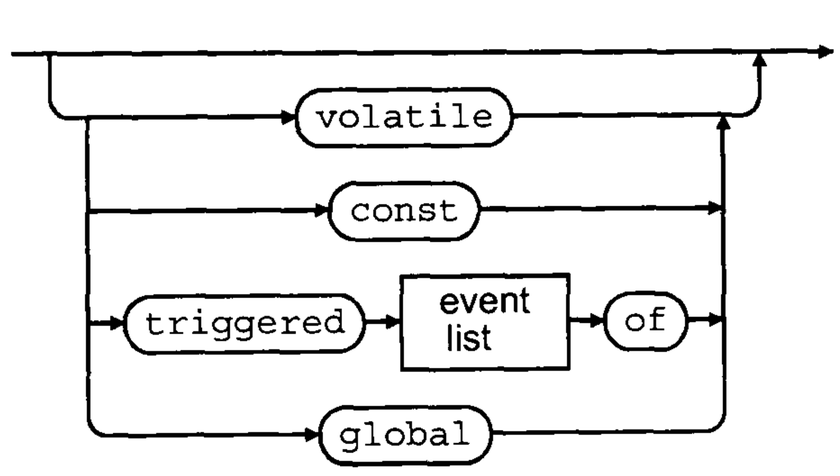
returntype:



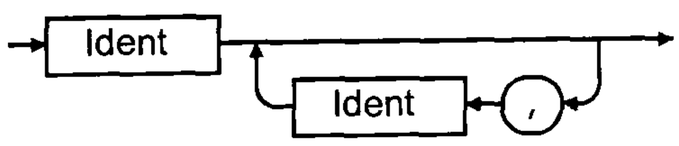
datatype:



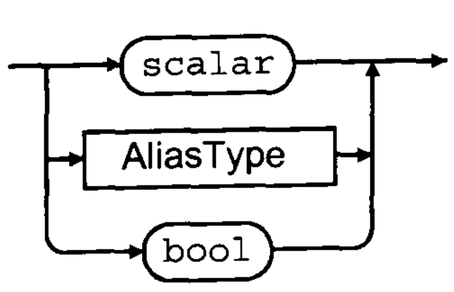
type-qualifier:



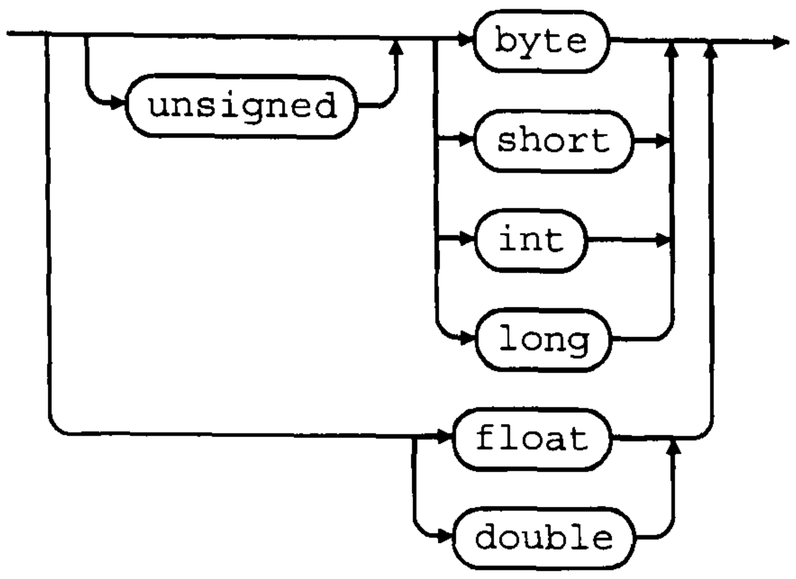
event-list:



type-primitive:

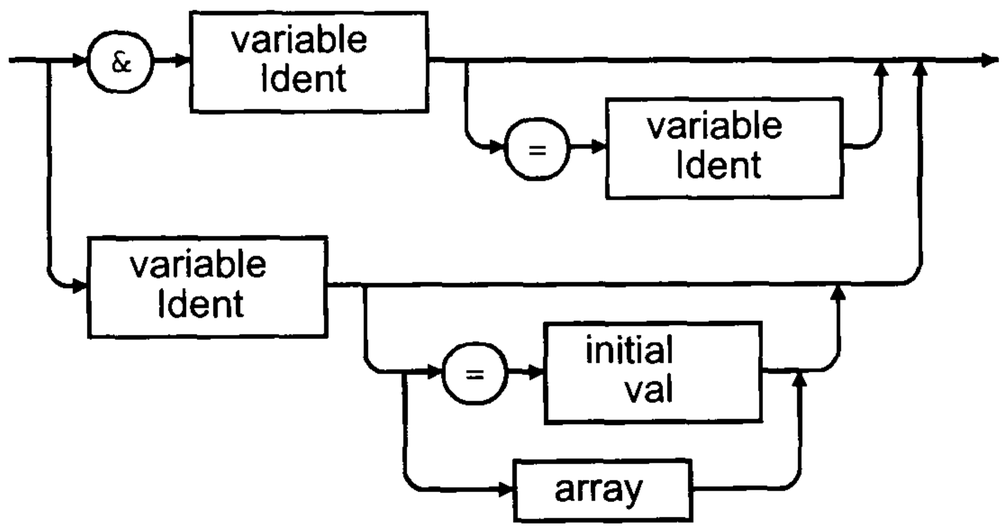


AliasType:

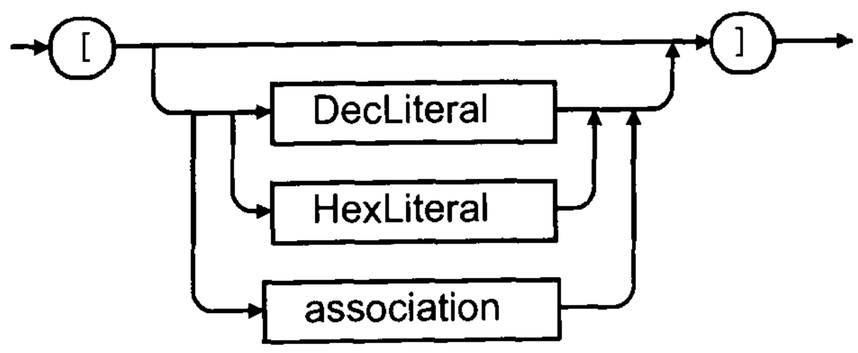


UserType:

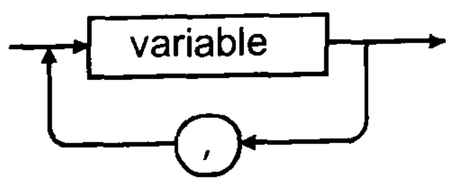
variable:



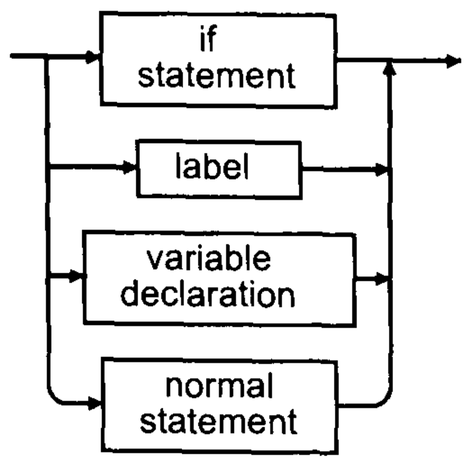
array:



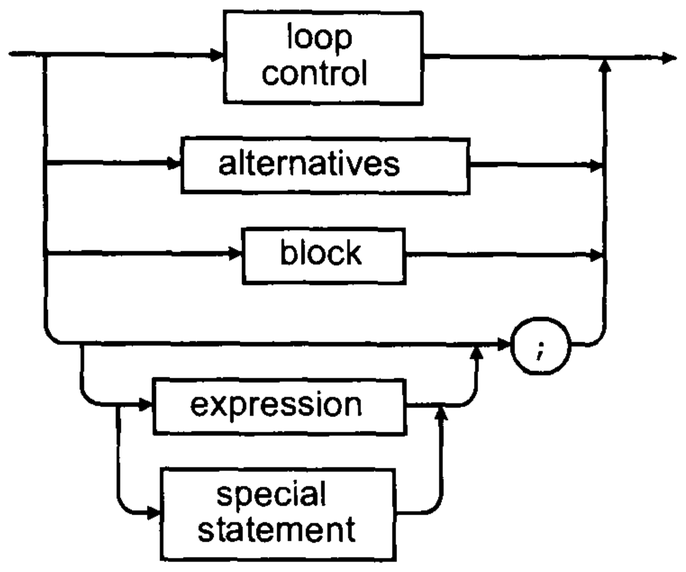
variable-list:



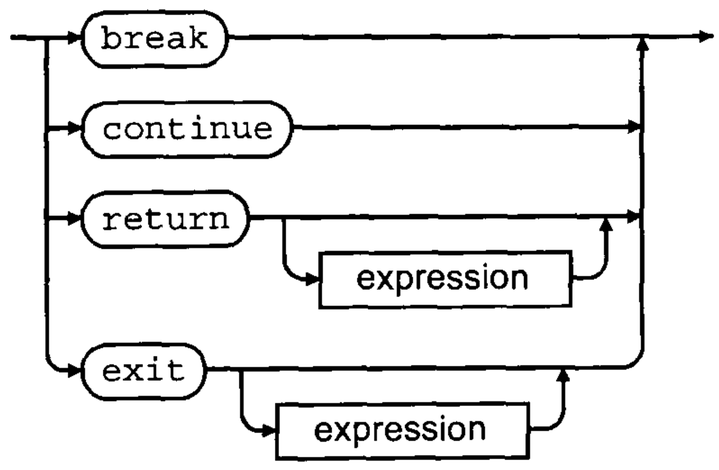
statement:



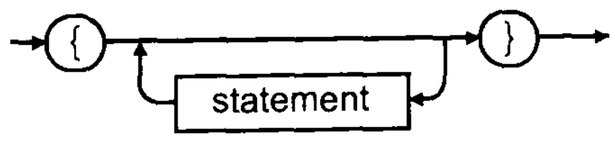
normal-statement:



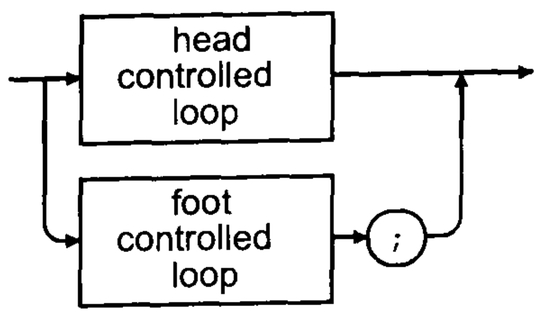
special-statement:



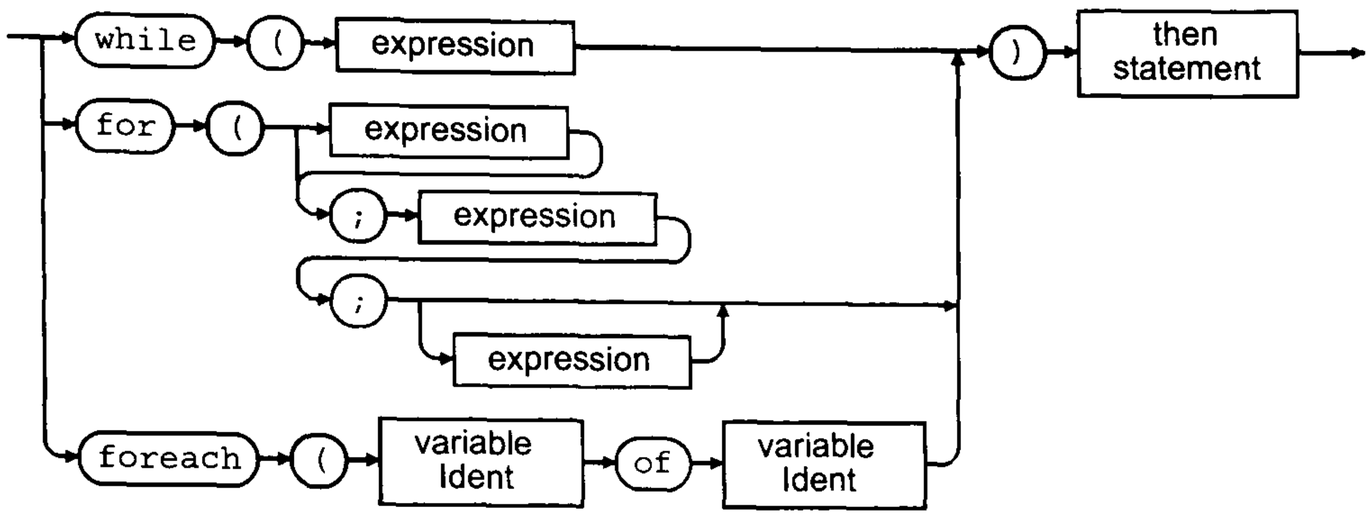
block:



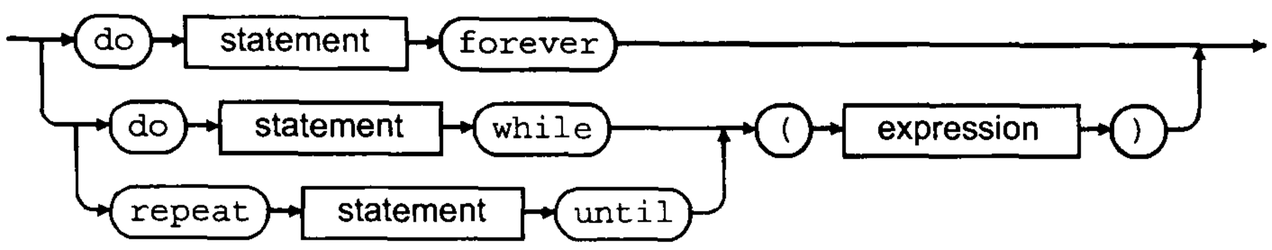
loop-control:



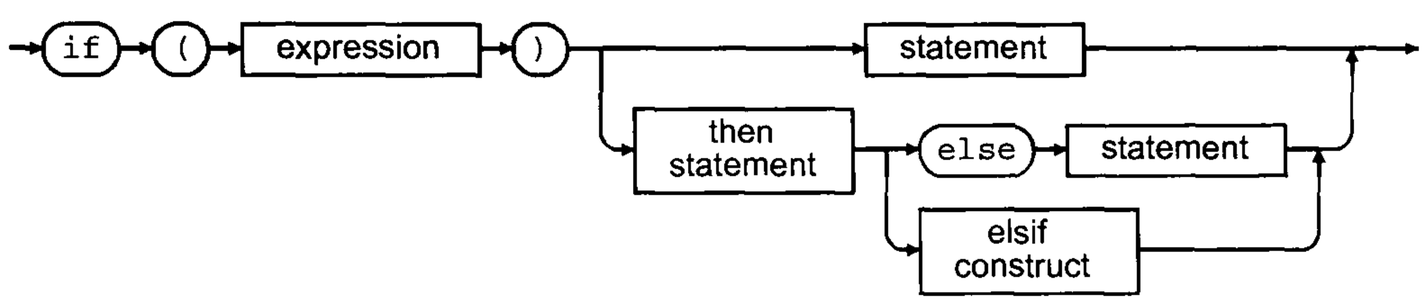
head-controlled-loop:



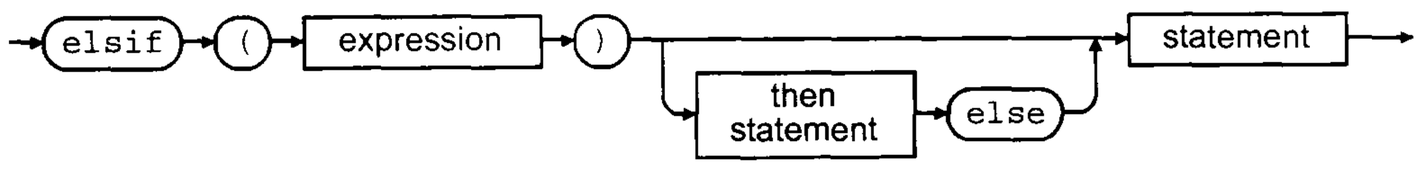
foot-controlled-loop:



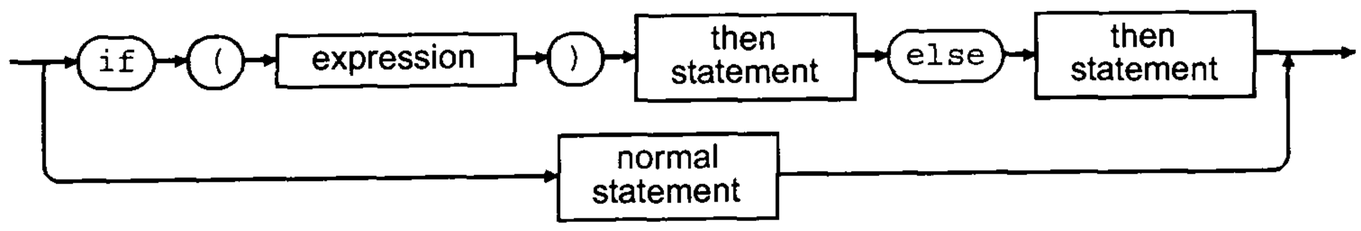
if-statement:



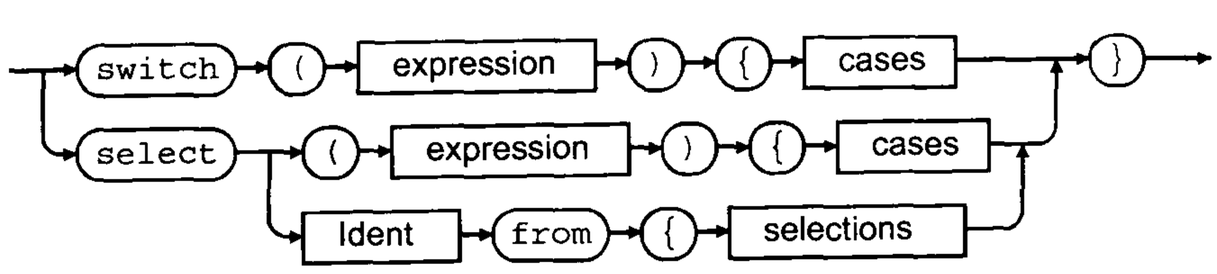
elsif-construct:



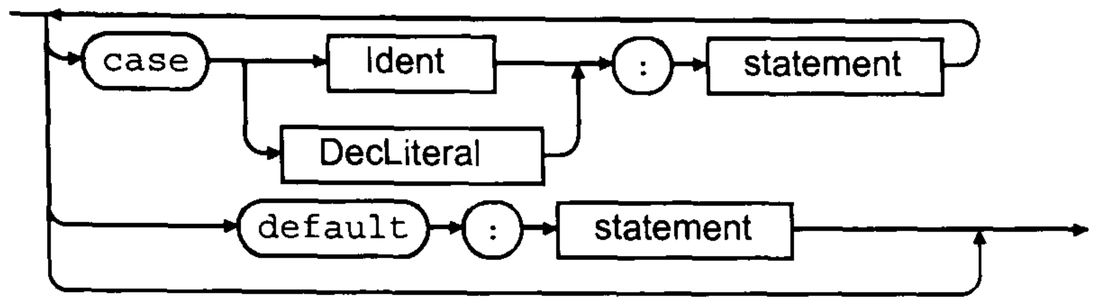
then-statement:



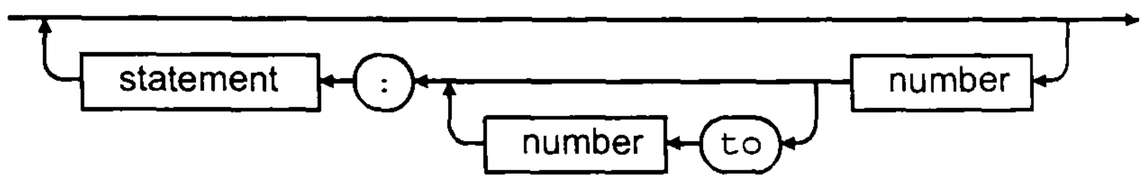
alternatives:



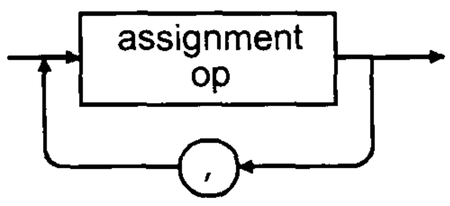
cases:



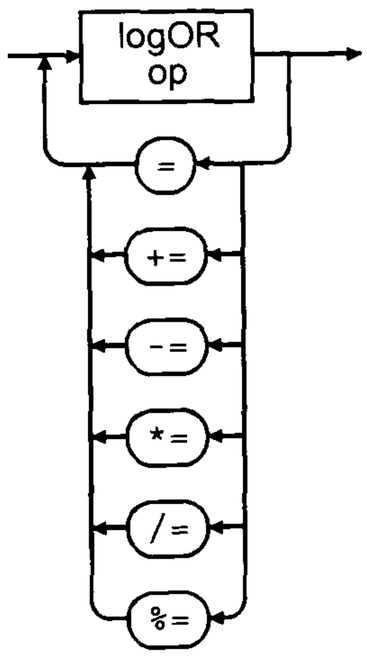
selections:



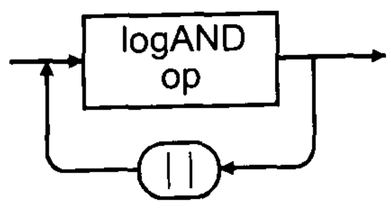
expression:



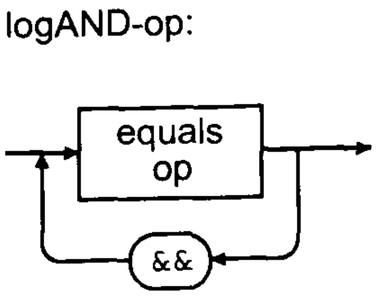
assignment-op:



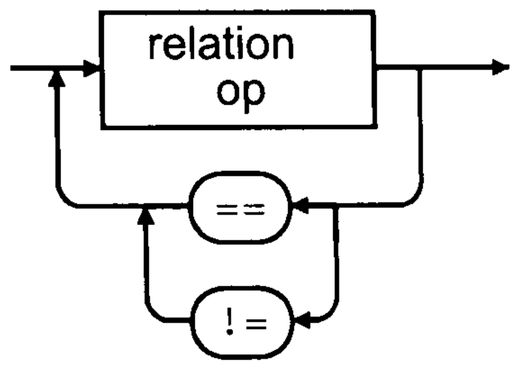
logOR-op:



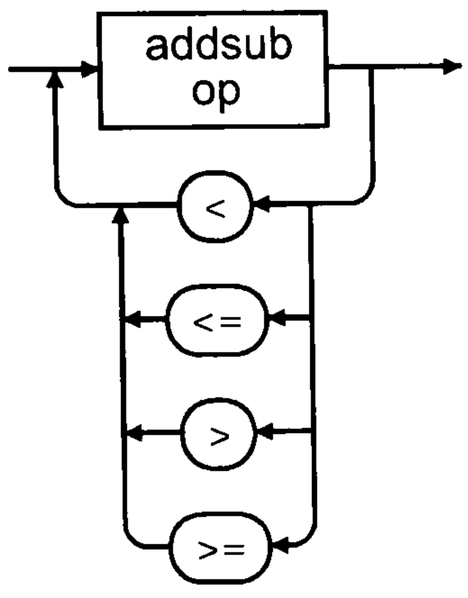
logAND-op:



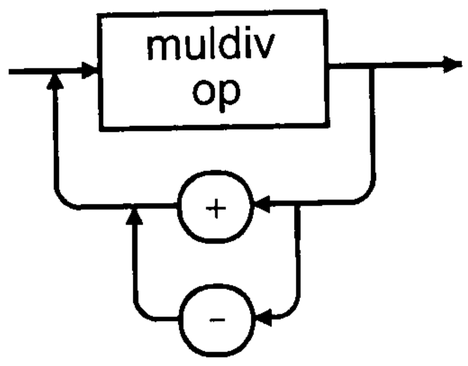
equals-op:



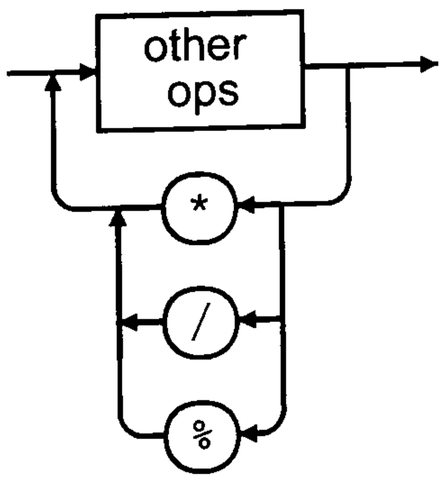
relation-op:



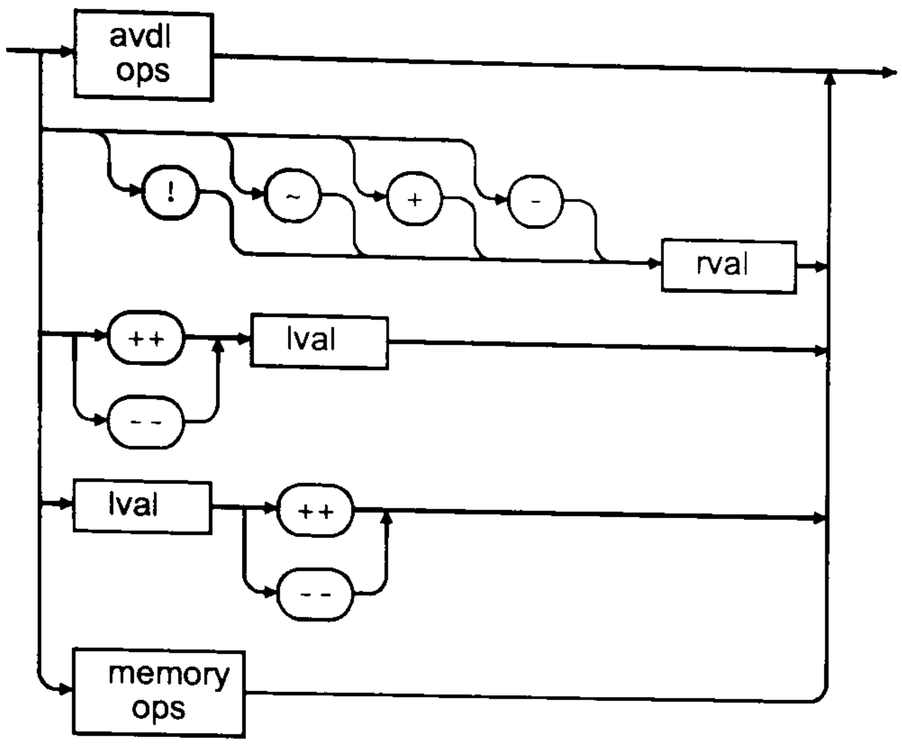
addsub-op:



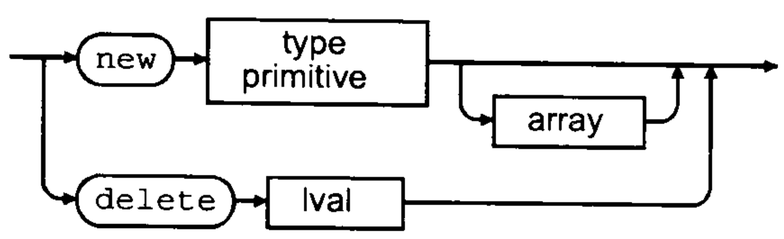
muldiv-op:



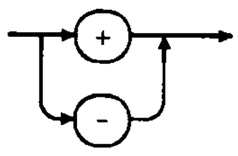
other-ops:



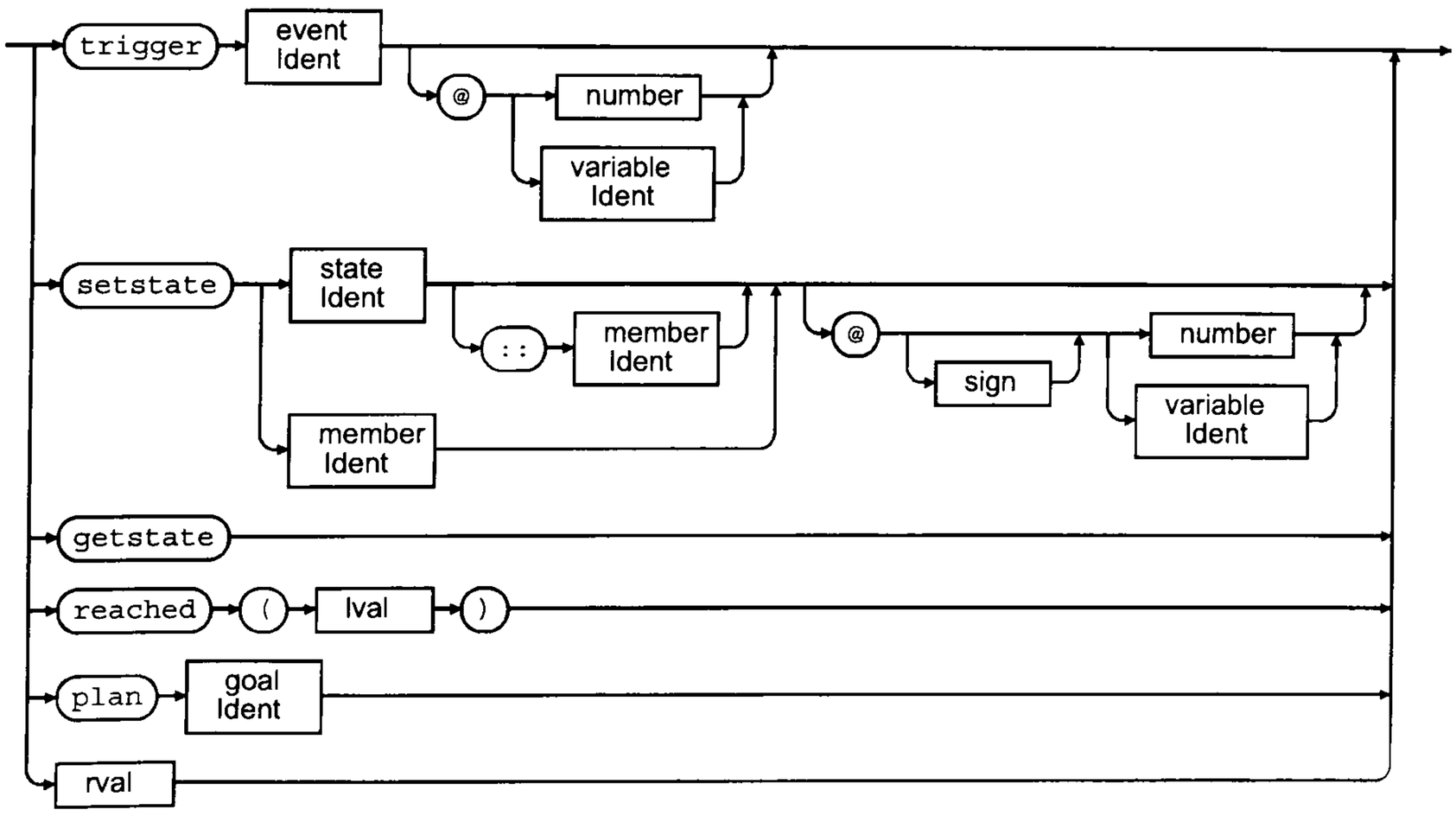
memory-ops:



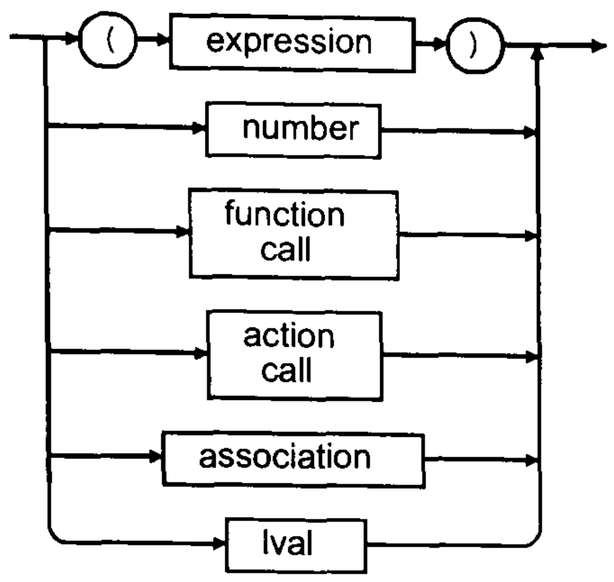
sign:



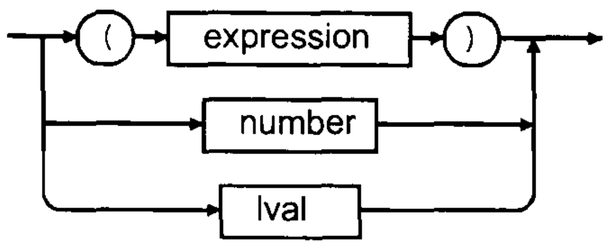
avdl-ops:



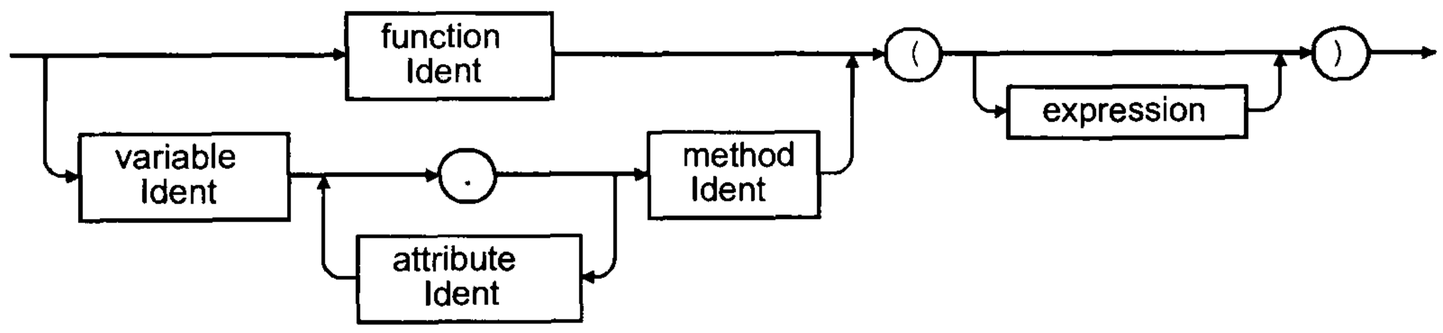
rval:



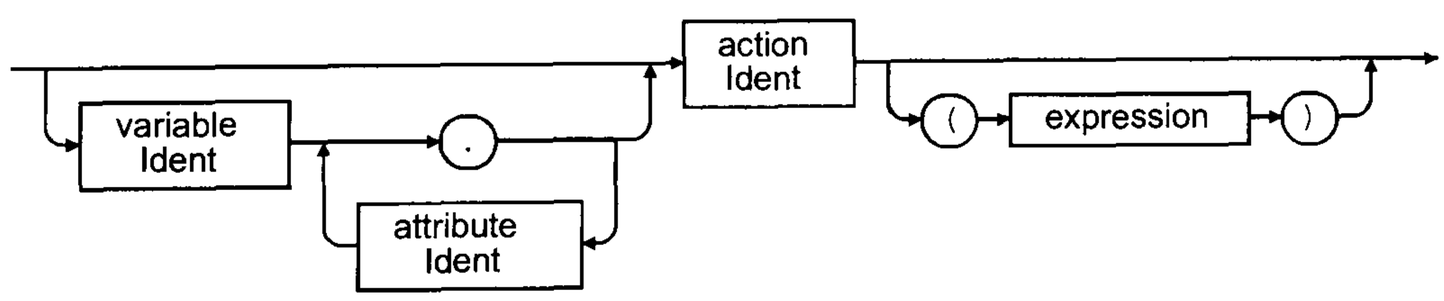
initial-val:



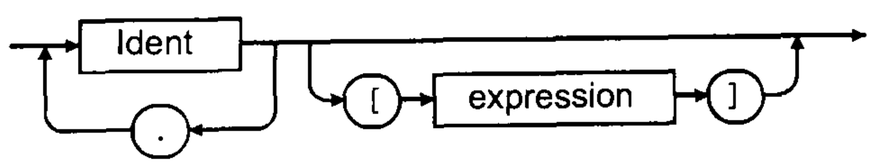
function-call:



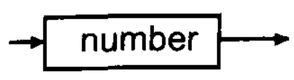
action-call:



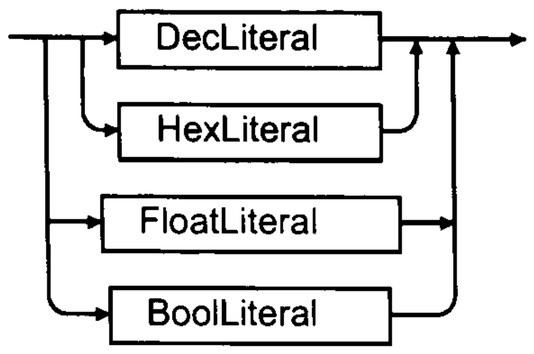
lval:



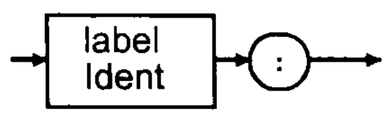
priority:



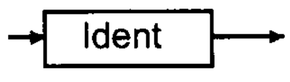
number:



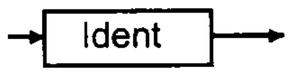
label:



association:



entity-Ident:



attribute-Ident:

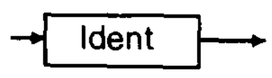
class-Ident:

function-Ident:

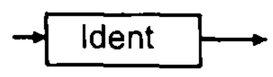
action-Ident:

goal-Ident:

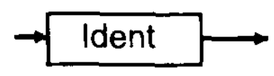
state-Ident:



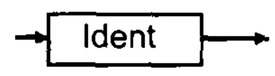
event-Ident:



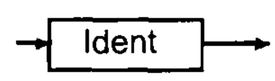
label-Ident:



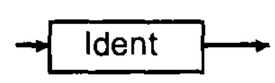
member-Ident:



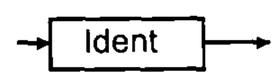
method-Ident:



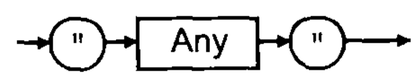
transition-Ident:



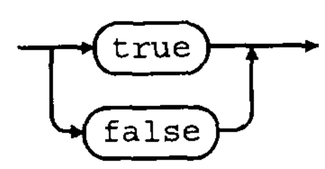
variable-Ident:



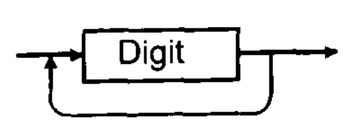
StringLiteral:



BoolLiteral:

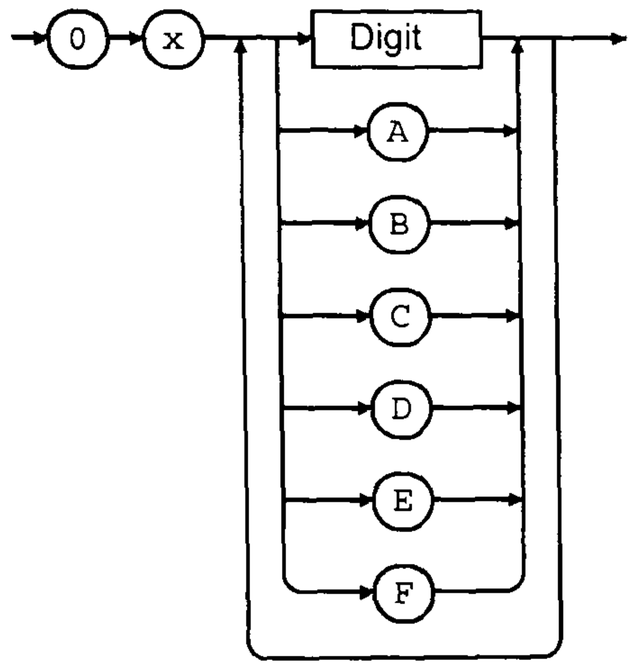


DecLiteral:

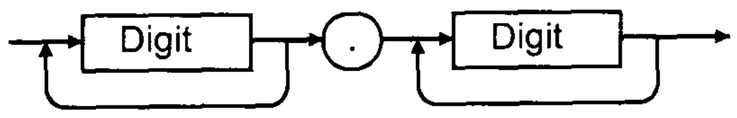


Any any printable ASCII character except "

HexLiteral:



FloatLiteral:



Digit any digit from '0' to '9'

Appendix E

The SEAL Scripting Language

This appendix presents the syntax of the SEAL scripting language that was described in Chapter 9 of this thesis. SEAL (Simple Entity Annotation Language) is a procedural scripting language for the definition of behaviour for virtual entities in computer games and the annotation of objects for use by these virtual entities. The language is a subset of AvDL (see Appendix D) that implements a substantial number of AvDL's features, the most significant of which are:

1. A finite state machine (FSM) data structure.

2. An event (trigger) data type.

3. Entity annotation.

The command syntax of SEAL is loosely based on the procedural production language C [Kerninghan and Ritchie 1988] (see Table E.1).

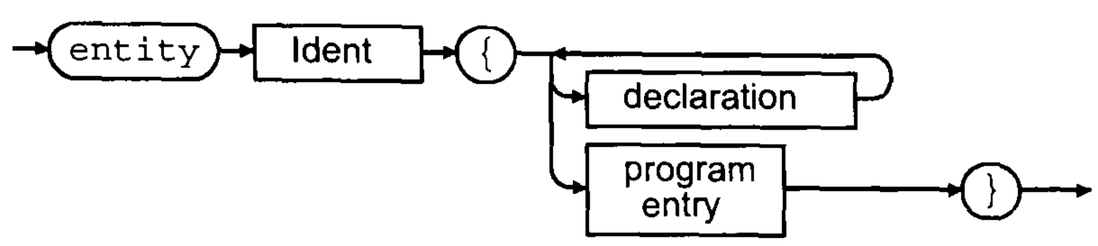
`action`	`break`	`case`	`const`	`continue`
`default`	`do`	`else`	`entity`	`event`
`exit`	`for`	`getstate`	`global`	`if`
`of`	`onentry`	`onexit`	`repeat`	`return`
`scalar`	`setstate`	`state`	`switch`	`trigger`
`triggered`	`until`	`while`	`void`	`volatile`

Table E.1: SEAL reserved words.

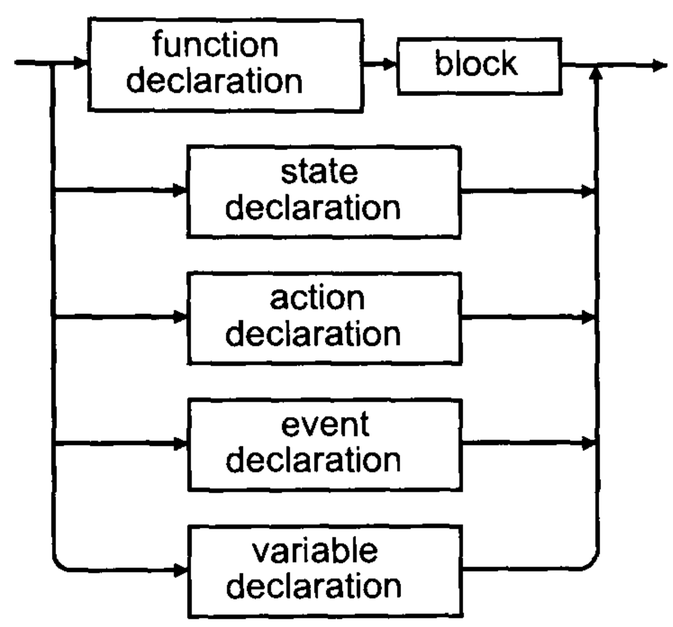
E.1 SEAL Syntax

program:

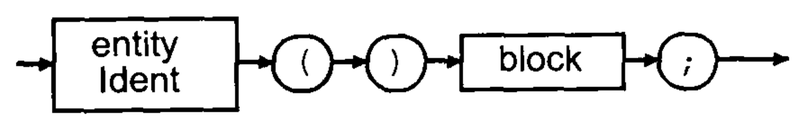
entity-declaration:



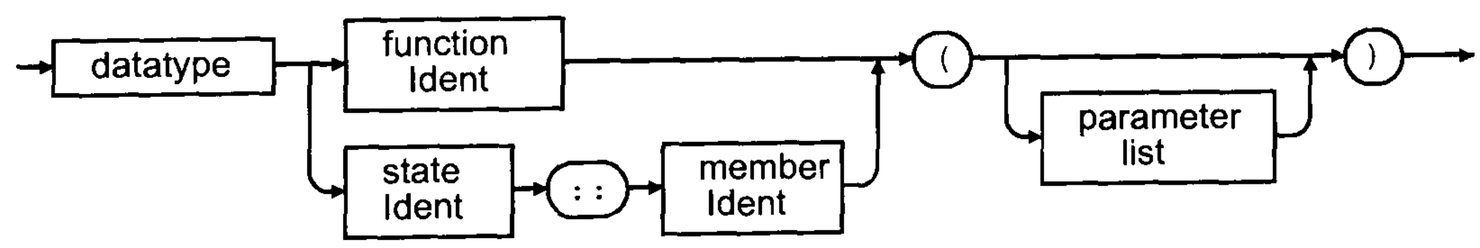
declaration:



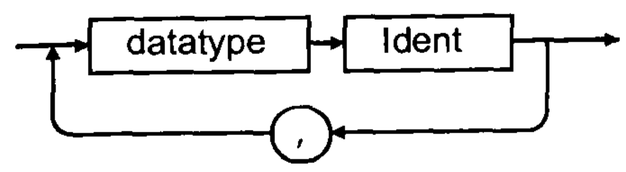
program-entry:



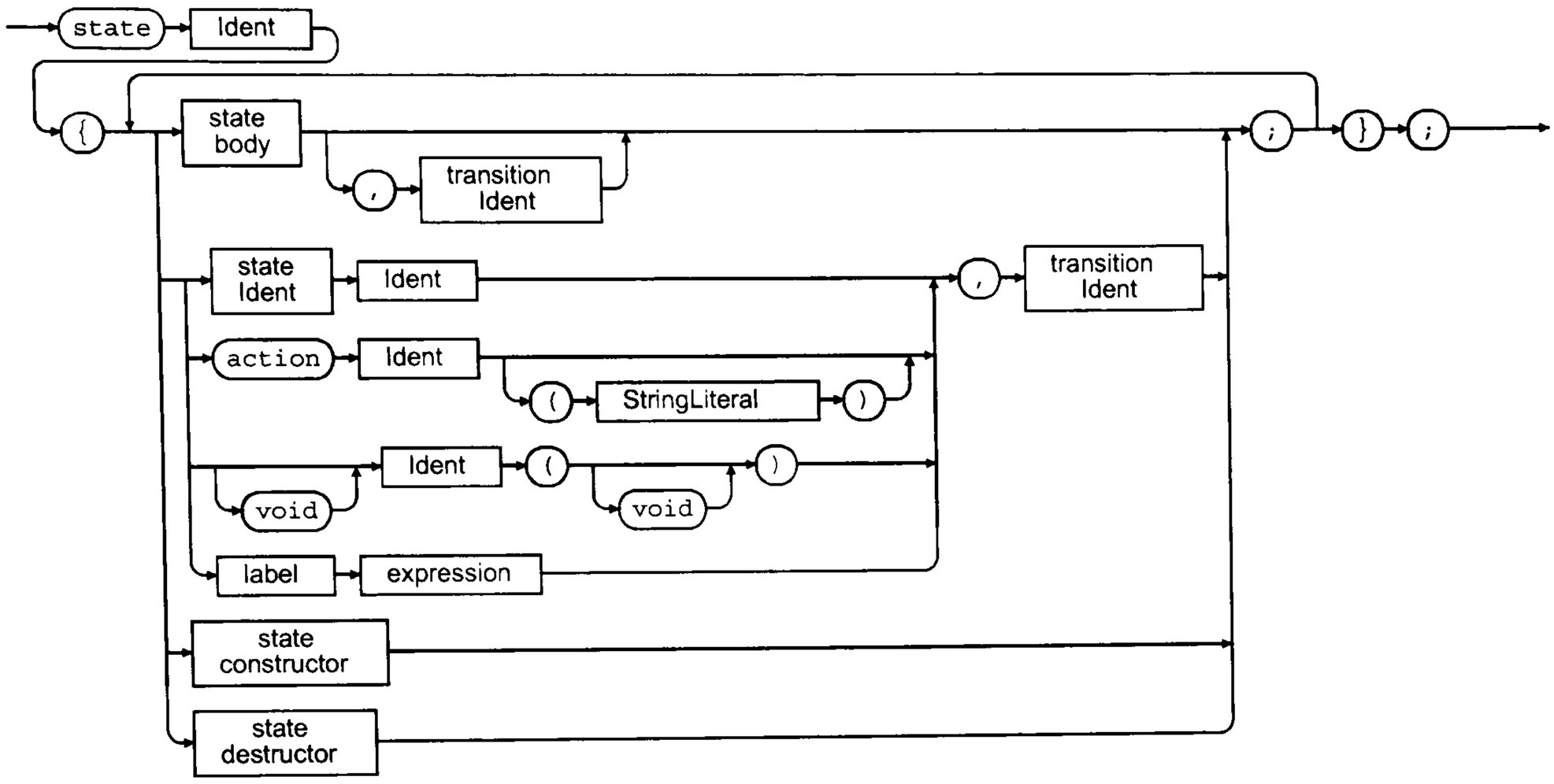
function-declaration:



parameter-list:

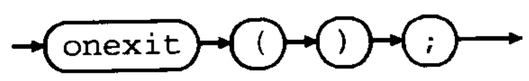


state-declaration:

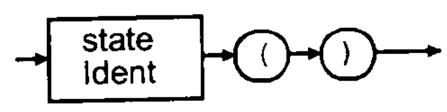


state-constructor:

state-destructor:

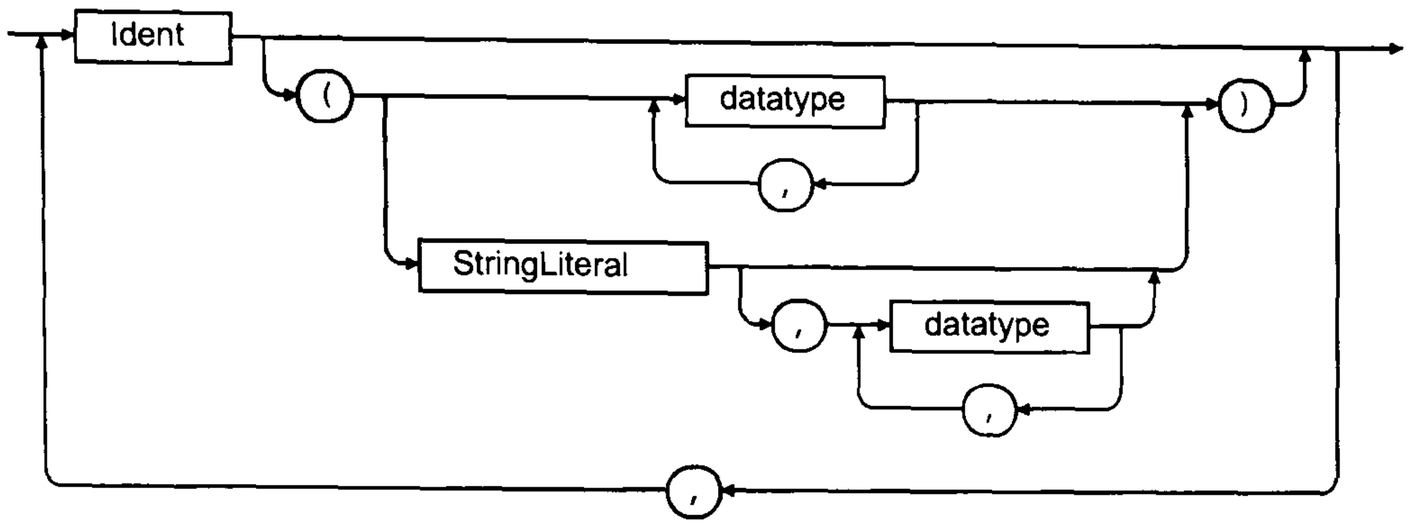


state-body:

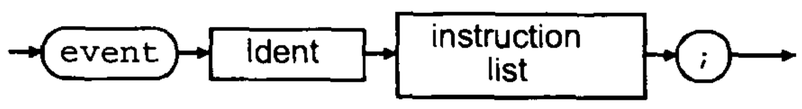


action-declaration:

action-list:



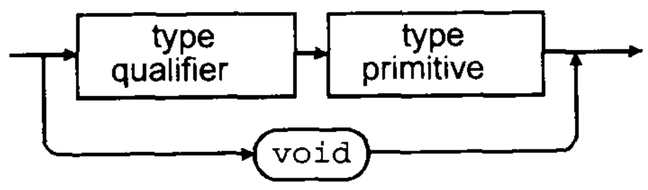
event-declaration:



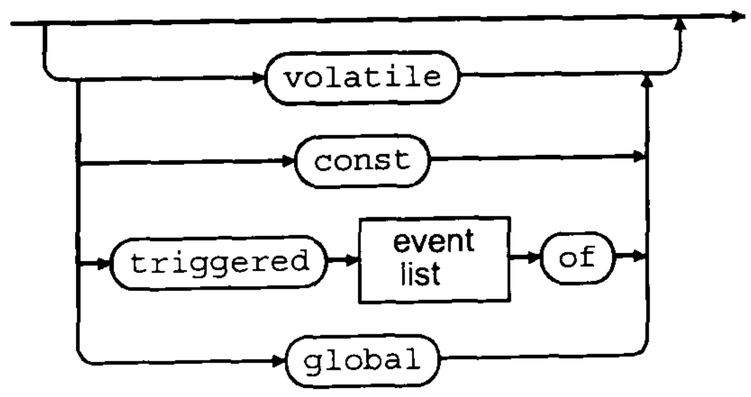
instruction-list:

variable-declaration:

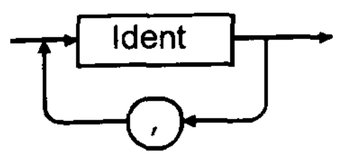
datatype:



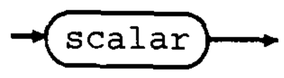
type-qualifier:



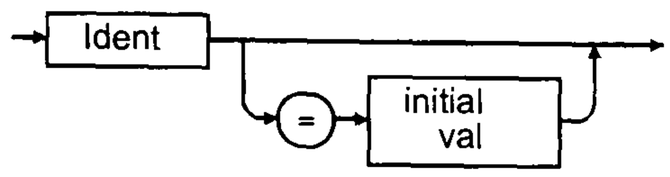
event-list:



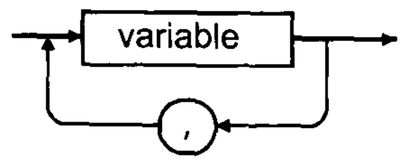
type-primitive:



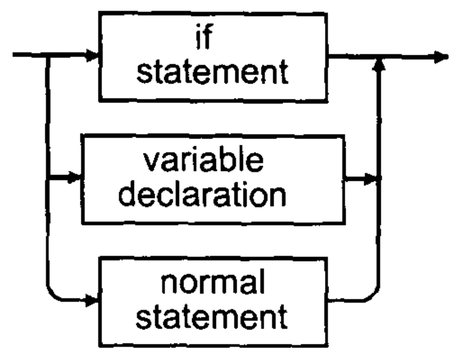
variable:



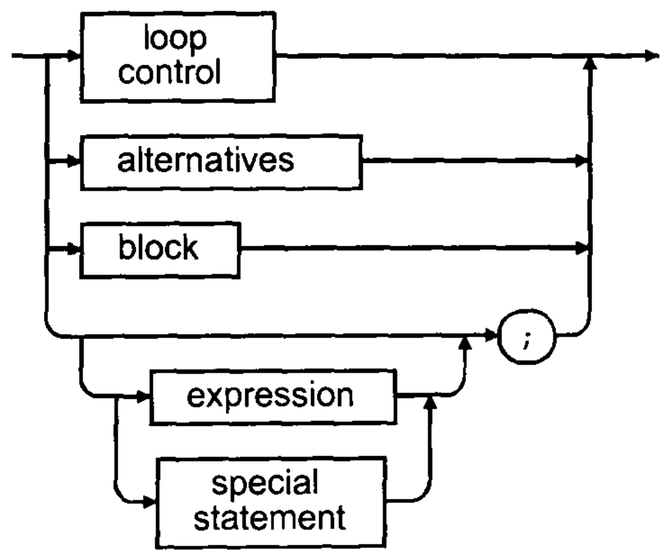
variable-list:



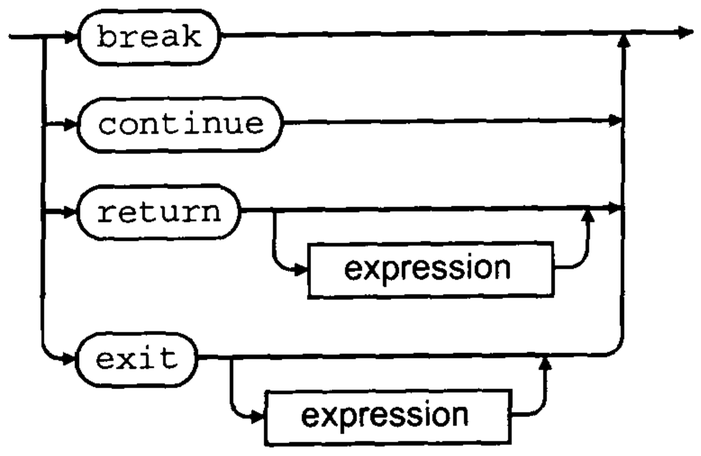
statement:



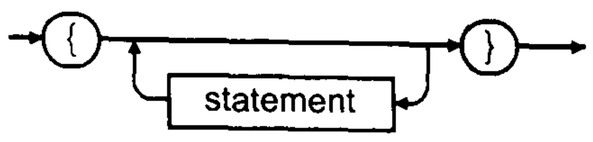
normal-statement:



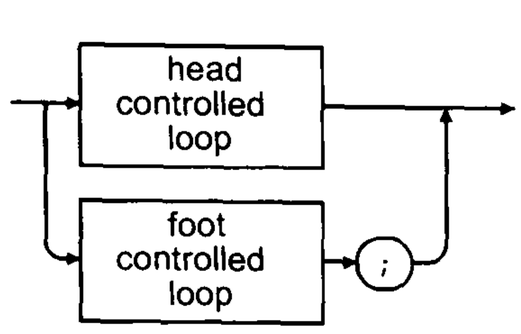
special-statement:



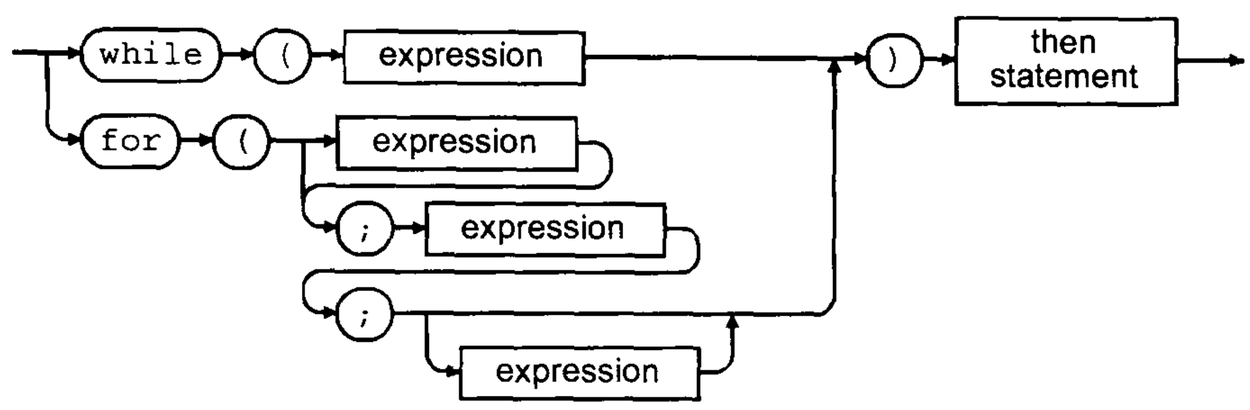
block:



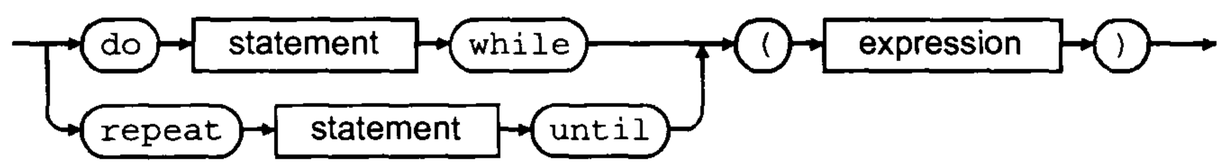
loop-control:



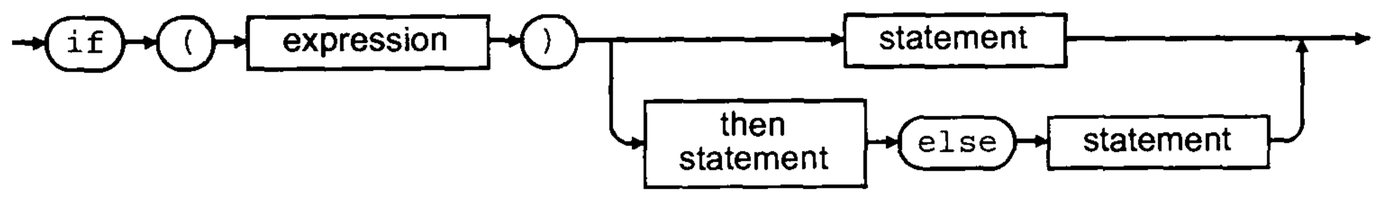
head-controlled-loop:



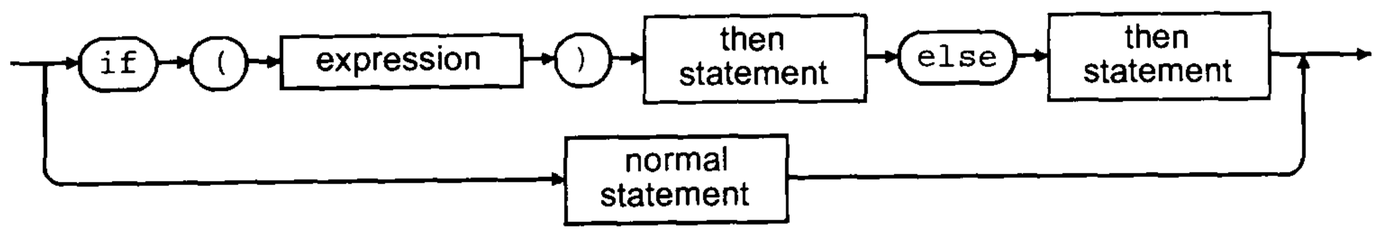
foot-controlled-loop:



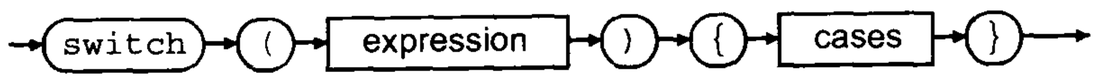
if-statement:



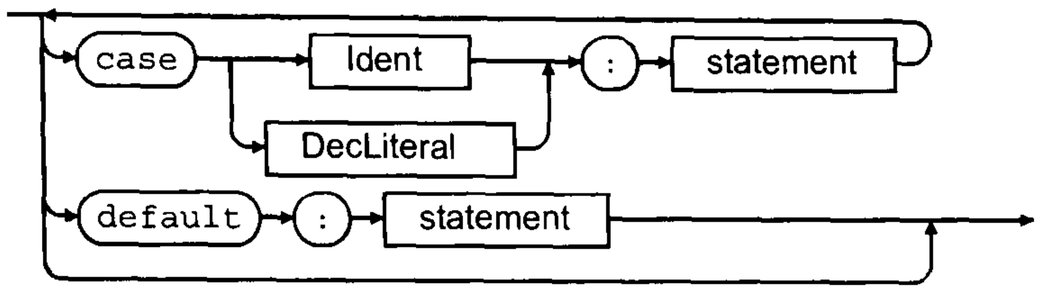
then-statement:



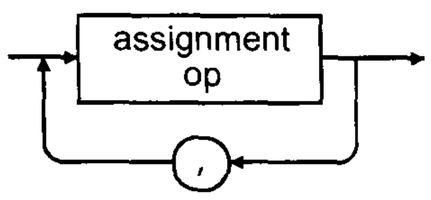
alternatives:



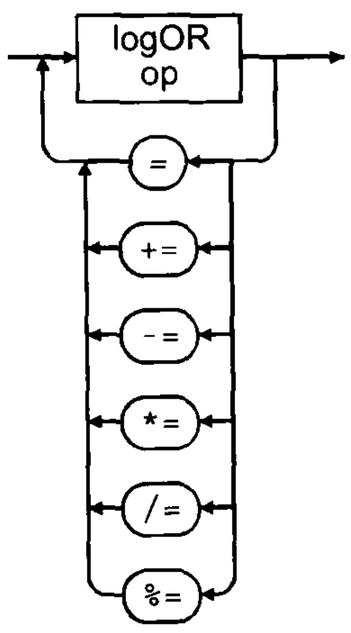
cases:



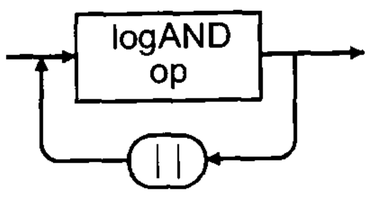
expression:



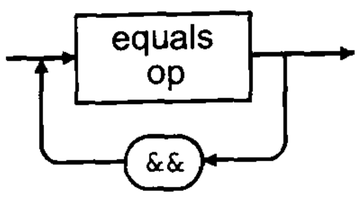
assignment-op:



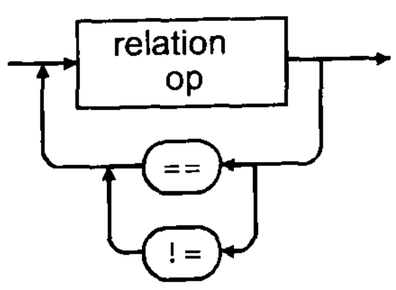
logOR-op:



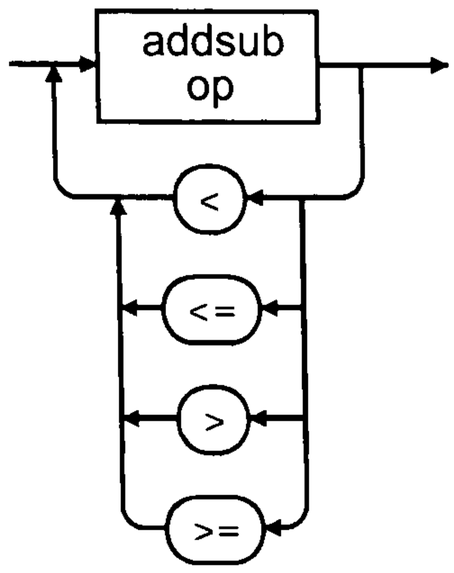
logAND-op:



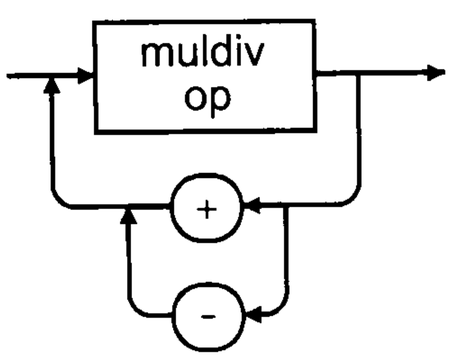
equals-op:



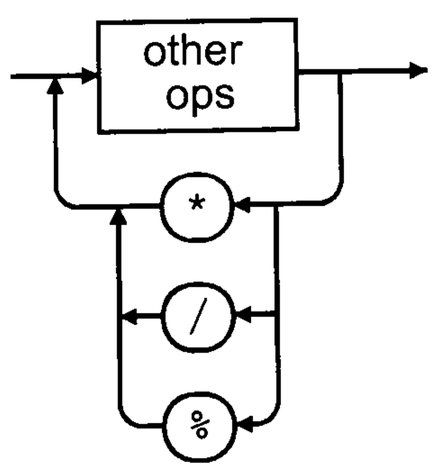
relation-op:



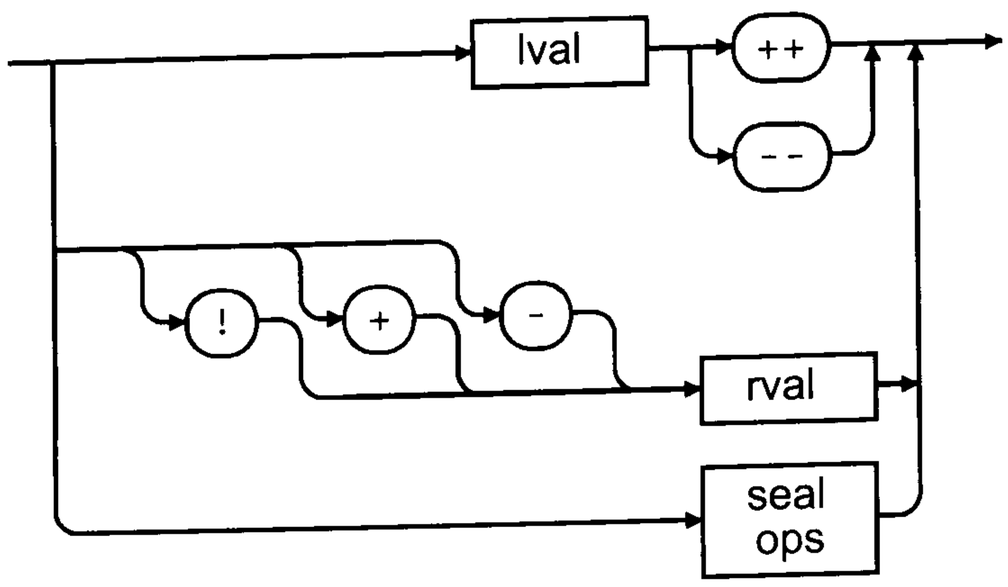
addsub-op:



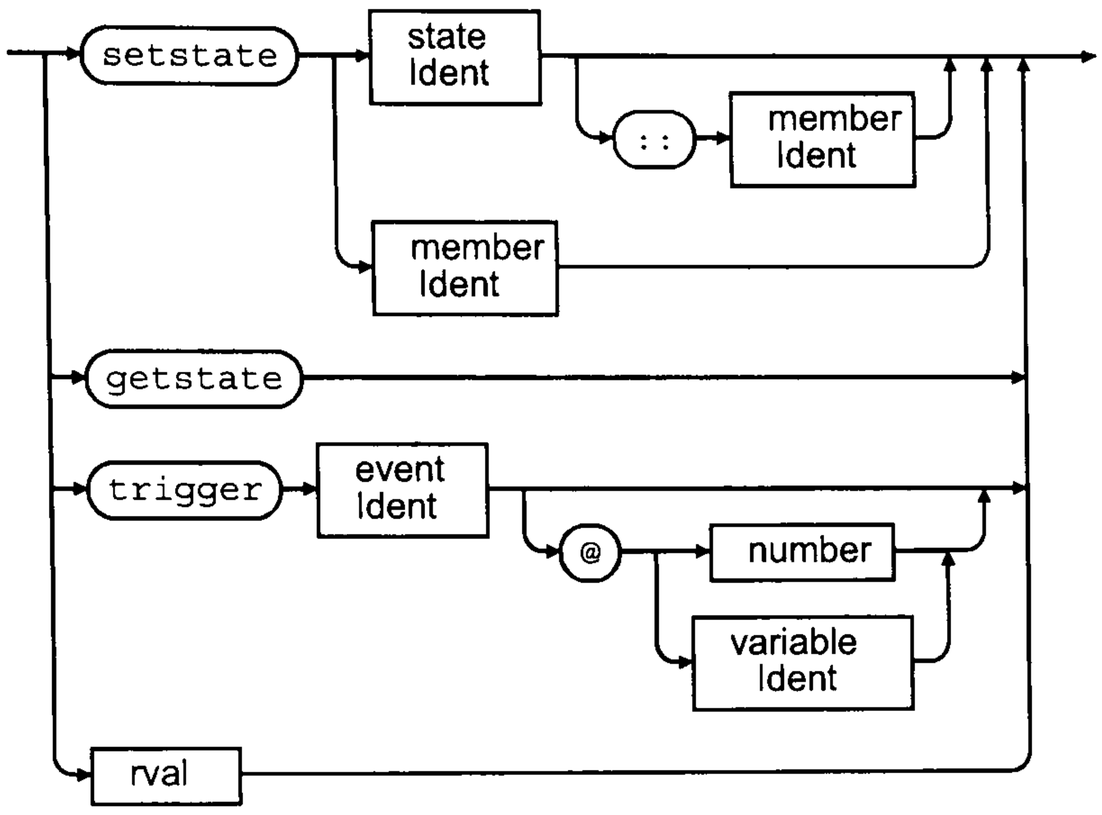
muldiv-op:



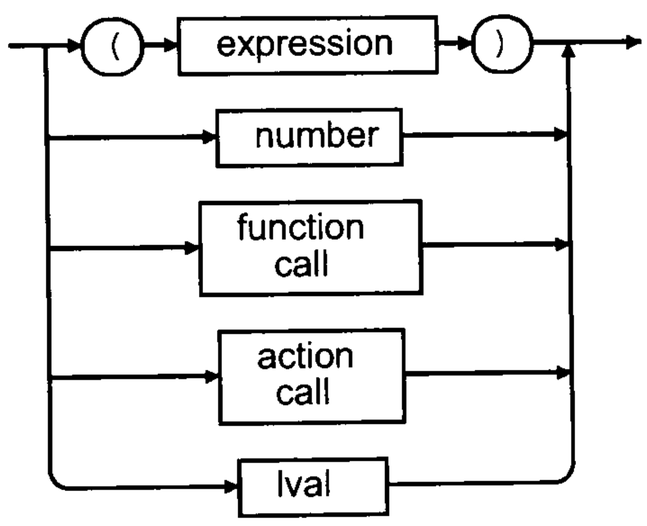
other-ops:



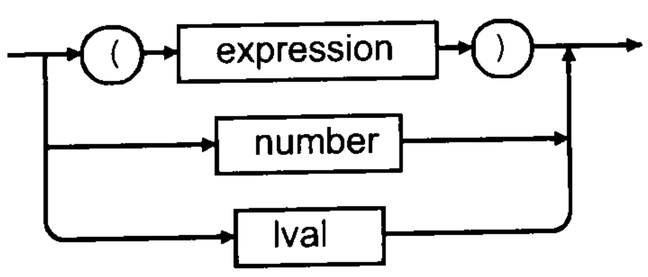
seal-ops:



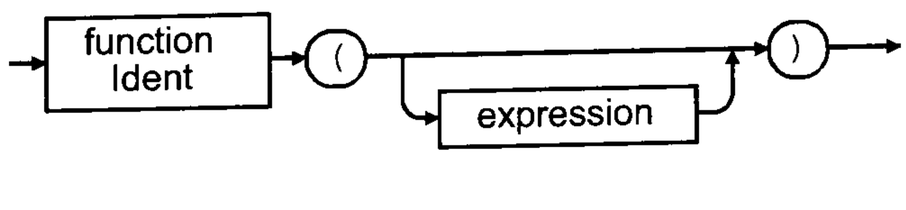
rval:



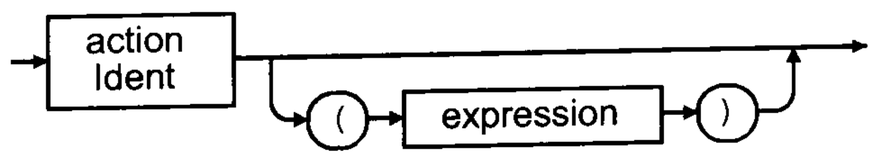
initial-val:



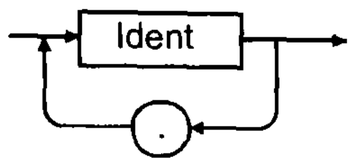
function-call:



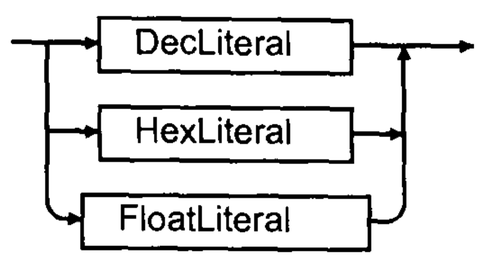
action-call:



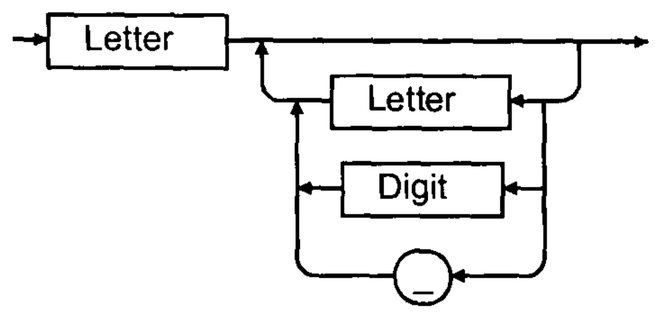
lval:



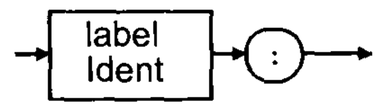
number:



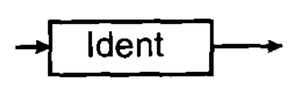
Ident:



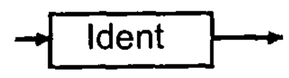
label:



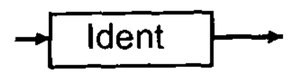
label-Ident:



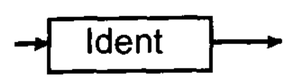
member-Ident:



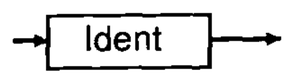
entity-Ident:



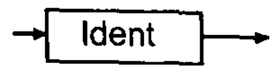
function-Ident:



action-Ident:

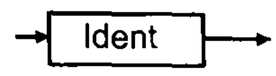


event-Ident:

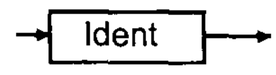


state-Ident:

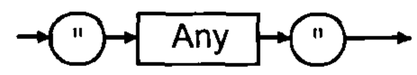
transition-Ident:



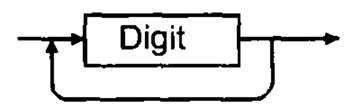
variable-Ident:



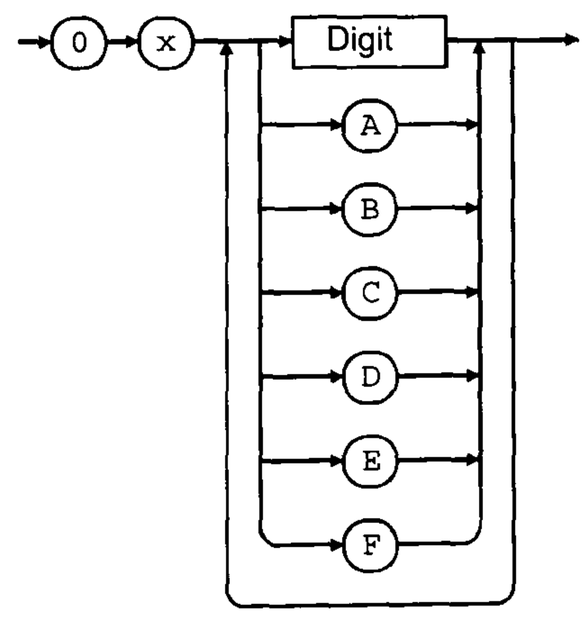
StringLiteral:



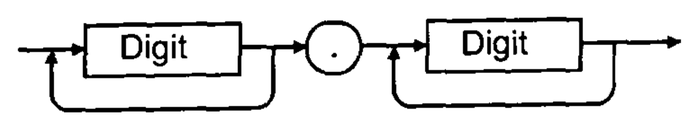
DecLiteral:



HexLiteral:



FloatLiteral:



Letter	any of the 26 letters of the alphabet (capital or lower case)
Digit	any digit from '0' to '9'
Any	any printable ASCII character except "

Appendix F

SEAL/AvDL System Prototype

Chapter 10 provided a discussion of the system prototype that implements the functionality of the SEAL subset of AvDL (see Chapter 9), as well as several additional features of AvDL that are not part of SEAL, such as AvDL's extension architecture. This appendix lists the system prototype's instruction set, several translation examples to demonstrate how the system is supposed to work, and a selection of API functions that show how the system can be integrated into a game engine.

F.1 Virtual Machine Instructions

The system prototype's virtual machine instructions, listed by their mnemonics.

F.1.1 Process Control Instructions

NOP – no operation
Placeholder instruction that does nothing.

SRT – start
Marks the entry point of the program.

STP – stop
Ends program execution.

XIT – exit
Exit command that reads the program's exit status from the process's stack.

HLD – hold process
Suspends the process until it is explicitly (re-)started by the virtual machine.

RST – reset
Resets the process to its initial state.

ISA – increment stack address
Increments the stack address register.

CLT – clear TOS (top of stack)
Removes the topmost element from the process's stack.

DSA – decrement stack address
Decrements the stack address register.

JMP – jump
Unconditional jump to a different instruction.

JPF – jump (if) false
Conditional jump to a different instruction.

ADM – allocate dynamic memory
Not yet implemented, reserved for future use.

FDM – free dynamic memory
Not yet implemented, reserved for future use.

F.1.2 Data Handling Instructions

LDC – load constant value
Loads a constant value onto the process's stack.

LOD – load variable
Loads the contents of a variable onto the process's stack.

LEX – load external (variable)
Not yet implemented.

STR – store variable
Stores the contents of the topmost entry on the process's stack into a variable.

LVA – load variable address
Retrieves a "pointer" to a variable.

LFA – load (data) from address
Dereference "pointer" and load data from the address onto the process's stack.

STA – store (data) to address
Dereference "pointer" and store data from the stack to the address.

LCS – load constant string
Loads a constant string into the process's memory.

LPA – load function address
Get a function "pointer". Not yet implemented - reserved for future use.

LDH – load high (segment)
Load high-segment value from variable onto the process's stack.

LDL – load low (segment)
Load low-segment value from variable onto the process's stack.

SRH – store high (segment)
Stores the contents of the topmost entry on the process's stack into the high-segment of a variable.

SRL – store low (segment)
Stores the contents of the topmost entry on the process's stack into the low-segment of a variable.

F.1.3 Function Handling Instructions

MES – mark exported (function) start
Marks the start of an exported function code segment.

MEE – mark exported (function) end
Marks the end of an exported function code segment.

CSF – call system function
Execute an intrinsic system function.

CUF – call user function
Calls a user-defined function.

CLX – call locally exported (function)
Calls a user-defined exported function within the current process.

CRX – call remote exported (function)
Calls a user-defined exported function that resides in a different process.

CEF – call extension function
Calls a function in an extension library.

BAR – block activation record
Creates a block activation record on the current process's stack.

RET – return
Returns program flow to the caller of the current function.

RFE – return from exported (function)
Returns program flow to the caller of the current exported function.

MHS – mark (event) handler start
Marks the start of an event handler code segment.

MHE – mark (event) handler end
Marks the end of an event handler code segment.

F.1.4 Comparisons

All comparisons compare the two topmost values on the stack and replace them with the result of the comparison.

EQU – equal
Compare if two values are equal.

NEQ – not equal
Compare if two values are not equal.

LES – less
Compare if the first value is less than the second value.

LEQ – less (or) equal
Compare if the first value is less than or equal to the second value.

GTR – greater
Compare if the first value is greater than the second value.

GEQ – greater (or) equal
Compare if the first value is greater than or equal to the second value.

F.1.5 Operators

All operations remove the operands from the stack and store the result on the stack.

NEG – negation
Unary arithmetic negation.

POW – power
Raises the first operand to the power of the second operand.

DIV – division
Arithmetic division.

MUL – multiplication
Arithmetic multiplication.

MOD – modulo
Results in the remainder of an arithmetic division (modulo).

ADD – add
Arithmetic addition.

SUB – subtract
Arithmetic subtraction.

INC – increment
Unary arithmetic increment by 1.0.

DEC – decrement
Unary arithmetic decrement by 1.0.

XOR – exclusive or
Logical exclusive "or".

IOR – inclusive or
Logical inclusive "or".

AND – and
Logical "and".

NOT – not
Unary logical negation.

PEQ – plus equals
Increment of a data value by the operand.

TEQ – times equals
Multiplication of a data value by the operand.

MEQ – minus equals
Decrement of a data value by the operand.

DEQ – div equals
Division of a data value by the operand.

REQ – remainder equals
Modulo of a data value divided by the operand.

ODD – odd
Unary test if a value is odd (or even).

XOP – extension operator
Apply extension operator. Not yet implemented – reserved for future use.

F.1.6 Heap Operations

These are not yet implemented, but reserved for future use.

LFH – load from heap
Loads the contents of a data entry from the heap onto the process's stack.

STH – store to heap
Stores the contents of the topmost entry on the process's stack into data entry on the heap.

LHA – load heap address
Load a "pointer" to an address on the heap.

LAH – load (from) address (on) heap
Dereference "pointer" and load data from the address on the heap onto the process's stack.

SAH – store (to) address (on) heap
Dereference "pointer" and store data from the stack to the address on the heap.

F.2 Intrinsic System Functions

executeCallback
System function that executes a callback function.

getExported
System function that retrieves a reference to an exported function whose location is unknown.

getFuncAddr
System function that retrieves a reference to an exported function from a known entity process.

retrievePID
System function that retrieves the process ID of the current entity process.

setBroadcast
System function that asks the virtual machine to advertise the process's exported functions.

setSilent
System function that asks the virtual machine to stop advertising the process's exported functions.

spawnEvent
System function that allows an entity process to trigger an event in the virtual machine.

stateTransition
System function that sets a process flag to trigger a state transition at the execution of the next instruction.

F.3 FSM Translation Example

AvDL/SEAL source code

```
action arm();
action unarm();
action attack();

triggered scalar of enemyDetected;
```

virtual machine instructions

setstate operator:

```
isa 3 # implicit set state function
lod 0  -1 # get parameter (next state)
str 1   4 # set next state
csf stateTransition # system function
lod 0  -1 # get parameter (next state)
ret 1 $1 # end function, return value
```

```
state guarding
{
    guarding();
};

state defending
{
    onentry();
    onexit();
    defending(), guarding;
};
```

```
guarding::guarding()
{
    do
    {
        if(enemyDetected)
            setstate defending;
    } forever;
}
```

finite state machine:

initialisation

```
ldc 1 # onentry
str 0  3 # flow target
ldc NULL
lod 1  3 # current state
neq # true (NULL) if FSM initialisation
jpf +10 # else run
ldc 100 # guarding
lod 1  4 # next state
neq
jpf +10 # initialise to "guarding"
ldc 200 # defending
lod 1  4 # next state
neq
jpf +49 # initialise to "defending"
jmp +88
```

FSM structure

```
ldc 100 # guarding
lod 1  3 # current state
equ
jpf +40 # current state is not "guarding"
```

state “guarding”

```
ldc 1 # onentry
lod 0  3 # flow target
equ
jpf +9
```

entry function (guarding)

```
lod 1  4 # next state
str 1  3 # current state
ldc NULL # transition target
```

```
str 1  4 # next state
ldc 2 # next: "guarding" body
str 0  3 # flow target
jmp -14 # continue
jmp +27 # never happens
ldc 2 # "guarding" body?
lod 0  3 # flow target
equ
jpf +12
```

state body (guarding)

```
ldc 1 # while(1)
jpf +6 # never happens
lod 1  8 # "enemyDetected"
jpf +3
ldc 200
cuf 1  2 $1 # call set state function
jmp -6 # loop back (while)
ldc 3 # onexit
str 0  3 # set next event
jmp -29 # continue
jmp +12 # never happens
ldc 3 # onexit?
lod 0  3 # flow target
equ
jpf +8
```

exit function (guarding)

```
lod 1  4 # next state
ldc NULL
neq
jpf +3
lod 1  4 # next state
cuf 1  2 $1 # call set state function
jmp +46
jmp +43
```

```
defending::onentry()
{
    arm();
}
```

```
defending::defending()
{
    attack();
}
```

```
ldc 200 # defending
lod 1  3 # current state
equ
jpf +39 # current state is not "defending"
```

state "defending"

```
ldc 1 # onentry
lod 0  3 # flow target
equ
jpf +11
```

entry function (defending)

```
lod 1  4 # next state
str 1  3 # current state
ldc 100 # transition target
str 1  4 # next state
lcs "arm"
csf executeCallback # system function
ldc 2 # next: "defending" body
str 0  3 # flow target
jmp -59 # continue
jmp +25 # never happens
ldc 2 # "defending" body?
lod 0  3 # flow target
equ
jpf +7
```

state body (defending)

```
lcs "attack"
csf executeCallback # system function
ldc 3 # onexit
str 0  3 # flow target
jmp -69 # continue
jmp +15 # never happens
ldc 3 # onexit
```

```
lod 0  3 # flow target
equ
jpf +11

exit function (defending)
lcs "unarm"
csf executeCallback # system function
lod 1  4 # next state
ldc NULL
neq
jpf +3
lod 1  4 # next state
cuf 1  2 $1 # call set state function
jmp +3

cleanup
jmp +2 # terminate FSM
jmp -86 # loop back
ret # return to main program
```

```
defending::onexit()
{
    unarm();
}

...
```

F.4 API Functions (Selection)

The API of the system prototype's run-time environment (selection).

F.4.1 Virtual Machine Control Functions

```
vm* Instance(void);
```
Returns a reference to the virtual machine.

```
int addProgram(std::string name);
```
Loads a SEAL bytecode program into a new entity process. Returns the ID of the new process.

```
bool registerEntity(int pID,entity *ve);
```

Register an object that was derived from the entity type with a process of the given ID.

`bool registerCallback(`<4 variations>`);`
Register a callback function that may be part of an entity object with the virtual machine or a given process, stating the name and ID of the callback function, whether it returns a value and the number of its formal parameters.

`bool registerEvent(std::string name);`
Register an event with the given name with the virtual machine. Returns the ID of the event.

`bool run(void);`
Execute the virtual machine's execution cycle.

F.4.2 Process Interaction Functions

`int getActiveProcesses(void);`
Returns the number of currently active entity processes.

`int getPriority(int pID);`
Returns the priority of the process with the given ID.

`void setPriority(int pID,int pr);`
Sets the priority of the process with the given ID to the stated value.

`bool isSuspended(int pID);`
Determines if the process with the given ID has been suspended.

`void spawnEvent(`<several variations>`);`
Triggers the event with the given ID or name. Optionally allows the specification of a target process.

`bool setValue(`<several variations>`);`

Set a named variable in an entity process to the given value. Optionally allows the specification of a target process.

F.4.3 Housekeeping Functions

`double getVersion(void);`
Return the version (number) of the virtual machine.

`double getRevisionNo(void);`
Return the revision (build) number of the virtual machine.

Glossary

AOT An AOT or Ahead-Of-Time compiler in a virtual machine is a compiler that first compiles a whole program into an intermediate form before it is executed by the virtual machine.

API An API or Application Programming Interface provides the programmer with an interface to a group of related functions that are usually located within a library of functions. The interface in this case is the description of data types, return types and formal parameters to functions and methods (if object orientation is used).

BDL A BDL or Behaviour Definition Language is a programming language used for the definition of game character behaviour. It facilitates the application oriented creation of believable virtual entities that inhabit game worlds.

DSL A DSL or Domain Specific Language is a programming language specialised for the purpose of solving problems in a specific application domain.

FPS An FPS or First Person Shooter game is an action video game in which the player experiences the gameplay from the viewpoint of the protagonist. This type of game usually involves the exploration of some sort of building complex and frequent skirmishes with other players or NPCs. Falise [2000] presents a study of the FPS game genre, providing an overview of its history.

FSM An FSM or Finite State Machine is a data structure that provides the most commonly used means for creating game AI [Fu and Houlette 2004]. In games, FSMs allow the definition of Boolean states of which only one will be active at any one time. Each state may have several possible follow states.

FuSM An FuSM or Fuzzy State Machine is a permutation of an FSM which uses fuzzy logic instead of Boolean logic [McCusky 2000].

GOAP GOAP or Goal-Oriented Action Planning is a goal oriented AI technique [Orkin 2004a] for use with virtual entities in which the sequence of actions that the system needs to perform to reach its end-state or goal is generated in real-time by using a planning heuristic on a set of known values which need to exist within the virtual entitys domain knowledge.

GP GP or Genetic Programming is an automated technique which produces algorithms by using a process that parallels evolution through natural selection, i.e. a simulation of life [Koza 1992].

GPU A GPU or Graphics Processing Unit is a co-processor with dedicated instructions for computer graphics operations. GPUs provide the functionality for modern computer graphics cards.

NPC An NPC or Non-Player Character is a virtual entity inhabiting the game world, whose perception and actions within the game are controlled by a computer program. The behaviour displayed by an NPCs is usually generated with the aid of "artificial intelligence" algorithms and techniques.

OTF An OTF or On-The-Fly compiler in a virtual machine is a compiler that compiles each instruction of a program immediately before it is executed by the virtual machine. The compilation target can be an intermediate form for use by the virtual machine or native machine code of the host platform.

RPG An RPG or Role Playing Game belongs to a computer game genre that has been derived from traditional paper-based games and board games like the popular "Dungeons and Dragons". In these games the player usually controls a hero character or a party of hero characters and needs to solve a series of quests within a fantasy setting.

RTS An RTS or Real-Time Strategy game is a strategy game which is not played round-based but in real-time, i.e. all of a player's units and his opponents have to be directed/make choices on the fly, while all action takes place simultaneously.

SDK An SDK or Software Development Kit is a comprehensive set of domain specific programs, libraries and manuals that provides a software developer with all the required data and information for developing programs in the SDK's domain.

VM A VM or Virtual Machine is a program that emulates the functionality of a whole computer system. It provides applications with a level of abstraction above the actual hardware (and the operating system) of the computer.

List of Publications

* ANDERSON, E. F. 2002. Off-Line Evolution of Behaviour for Autonomous Agents in Real-Time Computer Games. In *Proceedings of Parallel Problem Solving from Nature - PPSN VII*, vol. 2439 of *LNCS*, 689–699.

* ANDERSON, E. F. 2003. Playing Smart – Artificial Intelligence in Computer Games. In *Proceedings of zfxCON03 Conference on Game Development.*

* ZERBST, S., DÜVEL, O. AND ANDERSON, E. 2003. *3D-Spieleprogrammierung.* Markt + Technik.

ANDERSON, E. F. 2004. KI-nder der Zukunft – Ein kurzer Über- und Ausblick auf die Entwicklung von künstlicher Intelligenz in Computerspielen. Available from *http://www.games-net.de.*

* ANDERSON, E. F. 2004. A NPC Behaviour Definition System for Use by Programmers and Designers. In *Proceedings of CGAIDE 2004 5th Game-On International Conference on Computer Games: Artificial Intelligence, Design and Education*, 203–207.

* ANDERSON, E. F. 2005. Scripting Behaviour – Towards a New Language for Making NPCs Act Intelligently. In *Proccedings of zfxCON05 2nd Conference on Game Development*, 46–56.

* ANDERSON, E. F. 2005. SEAL – A Simple Entity Annotation Language. In *Proccedings of zfxCON05 2nd Conference on Game Development*, 70–73.

ANDERSON, E. F. AND MCLOUGHLIN, L. 2006. C-Sheep: Controlling Entities in a 3D Virtual World as a Tool for Computer Science Education. Poster in *Proceedings of Future Play 2006*

MCLOUGHLIN, L AND ANDERSON, E. F. 2006. I See Sheep: A Practical Application of Game Rendering Techniques for Computer Science Education. Poster in *Proceedings of Future Play 2006*

* ANDERSON, E. F. AND MCLOUGHLIN, L. 2006. Do robots dream of virtual sheep: Rediscovering the karel the robot paradigm for the plug&play generation. In *Proceedings of the Fourth Game Design and Technology Workshop and Conference (GDTW 2006)*, 92–96.

ANDERSON, E. F. AND MCLOUGHLIN, L. 2007. Critters in the classroom: A 3D computer-game-like tool for teaching programming to computer animation students. In *ACM SIGGRAPH 2007 Educators Program.*

*Publications relevant to this thesis.

References

Aho, A. V., Kernighan, B. W. and Weinberger, P. J. 1979. Awk - a Pattern Scaning and Processing Language (Second Edition). *Software: Practice & Experience 9*(4), pp. 267–280.

Anderson, E. F. and McLoughlin, L. 2006. Do Robots Dream of Virtual Sheep: Rediscovering the Karel the Robot Paradigm for the Plug&Play Generation. In *Proceedings of the Fourth Game Design and Technology Workshop and Conference (GDTW 2006)*, pp. 92–96.

Anderson, E. F. and McLoughlin, L. 2007. Critters In The Classroom: A 3D Computer-Game-Like Tool for Teaching Programming to Computer Animation Students. In *ACM SIGGRAPH 2007 Educators Program.*

Anderson, E. F. 2002. Off-Line Evolution of Behaviour for Autonomous Agents in Real-Time Computer Games. In *Proceedings of Parallel Problem Solving from Nature - PPSN VII*, vol. 2439 of *LNCS*, pp. 689–699.

Anderson, E. F. 2003a. Playing Smart - Artificial Intelligence in Computer Games. In *Proceedings of zfxCON03 Conference on Game Development.*

Anderson, E. F. 2003b. ZBL/0 - the ZFX Bot Language - Specification. Available from: http://zbl0.zfx.info. [Accessed 29/02/2008].

Anderson, E. F. 2004. A NPC Behaviour Definition System for Use by Programmers and Designers. In *Proceedings of CGAIDE 2004 5th Game-On International Conference on Computer Games: Artificial Intelligence, Design and Education*, pp. 203–207.

Anderson, E. F. 2005a. Scripting Behaviour - Towards a New Language for Making NPCs Act Intelligently. In *Proccedings of zfxCON05 2nd Conference on Game Development*, pp. 46–56.

Anderson, E. F. 2005b. SEAL - a Simple Entity Annotation Language. In *Proccedings of zfxCON05 2nd Conference on Game Development*, pp. 70–73.

ATKIN, M. S., WESTBROOK, D. L. AND COHEN, P. 1999. Capture the Flag: Military Simulation Meets Computer Games. In *Proceedings of AAAI Spring Symposium Series on AI and Computer Games 1999*, pp. 1–5.

BABA, S. A., HUSSAIN, H. AND EMBI, Z. C. 2007. An Overview of Parameters of Game Engine. *IEEE Multidisciplinary Engineering Education Magazine 2*(3), pp. 10–12.

BEAUBOUEF, T. AND MASON, J. 2005. Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *ACM SIGCSE Bulletin 37*(2), pp. 103–106.

BERTRAND, F. AND AUGERAUD, M. 1999. BDL: A Specialized Language for Per-Object Reactive Control. *IEEE Transactions on Software-Engineering 25*(3), pp. 347–362.

BEZROUKOV, N. 2006. Scripting Languages as a Step in Evolution of Very High Level Languages. Available from: http://www.softpanorama.org/People/Scripting_giants/scripting_languages_as_vhll.shtml [Accessed 29/02/2008].

BILAS, S. 2002. *Dungeon Siege Technical Manual.* Available from: http://www.drizzle.com/ scottb/ds/skrit.htm [Accessed 29/02/2008].

BINSUBAIH, A., MADDOCK, S. AND ROMANO, D. 2007. A Survey of 'Game' Portability. Tech. Rep. CS-07-05, University of Sheffield. Department of Computer Science.

BLOW, J. 2002. Toward Better Scripting. *Game Developer 9*(10).

BLUNDEN, B. 2002. *Virtual Machine Design and Implementation.* Wordware.

BOER, J. 2000. Object-Oriented Programming and Design Techniques. In *Game Programming Gems.* Charles River Media, pp. 8–19.

BÖHM, C. AND JACOPINI, G. 1966. Flow Diagrams, Turing Machines and Languages with only Two Formation Rules. *Communications of the ACM 9*(5), pp. 366–371.

Boselli, L. 2004. GUN-TACTYX - Historical Background. Available from: http://guntactyx.gameprog.it/. [Accessed 29/02/2008].

Brockington, M. and Darrah, M. 2002. How Not to Implement a Basic Scripting Language. In *AI Game Programming Wisdom*. Charles River Media, pp. 548–544.

Brockington, M. 2002. Level-Of-Detail AI for a Large Role-Playing Game. In *AI Game Programming Wisdom*. Charles River Media, pp. 419–425.

Brom, C., Gemrot, J., Bda, M., Burkert, O., Partington, S. J. and Bryson, J. J. 2006. POSH Tools for Game Agent Development by Students and Non-Programmers. In *Proceedings of the 9th International Computer Games Conference: AI, Mobile, Educational and Serious Games (CGAMES 2006)*, pp. 126–133.

Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A. and Miller, P. 1997. Mini-languages: A Way to Learn Programming Principles. *Education and Information Technologies 2*(1), pp. 65–83.

Calloni, B. A. and Bagert, D. J. 1995. Iconic Programming for Teaching the First Year Programming Sequence. In *IEEE FIE '95: Proceedings of the Frontiers in Education Conference on 1995. Proceedings, 1995 vol 1.*, pp. 2a5.10–2a5.13.

Campbell, B. 2006. Swiss Army Chainsaw: A Common Sense Approach to Tool Development. Available from: http://www.gamasutra.com. [Accessed 29/02/2008].

Carlisle, P. 2002. Designing a GUI Tool to Aid in the Development of Finite-State Machines. In *AI Game Programming Wisdom*. Charles River Media, pp. 71–77.

Cass, S. 2002. Mind Games. *IEEE Spectrum 39*(12), pp. 40–44.

Champandard, A. J. 2004. *AI Game Development*. New Riders.

COHEN, M. A., RITTER, F. E. AND HAYNES, S. R. 2005. Herbal: A High-Level Language and Development Environment for Developing Cognitive Models in Soar. In *In Proceedings of the 14th Conference on Behaviour Representation in Modeling and Simulation*, pp. 133–140.

COLLINS. 2001a. Artificial Intelligence. *Collins English Dictionary*, fifth ed. HarperCollins.

COLLINS. 2001b. Avatar. *Collins English Dictionary*, fifth ed. HarperCollins.

COLMERAUER, A. AND ROUSSEL, P. 1993. The Birth of Prolog. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pp. 37–52.

COMBS, N. AND ARDOINT, J. 2004. Declarative versus Imperative Paradigms in Games AI. Available from: http://www.red3d.com/cwr/games/. [Accessed 29/02/2008].

CORADESCHI, S. AND SAFFIOTTI, A. 1999. Symbolic Object Descriptions to Sensor Data. Problem Statement. *Linköping Electronic Articles in Computer and Information Science 4*(9).

CORNWELL, J., O'BRIEN, K., SILVERMAN, B. AND TOTH, J. 2003. Affordance Theory for Improving the Rapid Generation, Composability, and Reusability of Synthetic Agents and Objects. In *BRIMS 2003: Proceedings of the Twelfth Conference on Behavior Representations in Modeling and Simulation.*

CUTIMITSU, M., SZAFRON, D., SCHAEFFER, J., MCNAUGHTON, M., ROY, T., ONUCZKO, C. AND CARBONARO, M. 2006. Generating Ambient Behaviors in Computer Role-Playing Games. *IEEE Intelligent Systems 21*(5), pp. 19–27.

DANC. 2006. Managing Game Design Risk: Part II – Data Driven Development. Blog Entry, available from: http://www.lostgarden.com. [Accessed 29/02/2008].

DANN, W., COOPER, S. AND PAUSCH, R. 2000. Making the Connection: Programming with Animated Small World. In *ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education*, pp. 41–44.

DARKEN, C. J. 2007. Level Annotation and Test by Autonomous Exploration. In *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2007)*.

DAVIS, B., BEATTY, A., CASEY, K., GREGG, D. AND WALDRON, J. 2003. The Case for Virtual Register Machines. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pp. 41–49.

DAWSON, B. 2001. Micro-Threads for Game Objects AI. In *Game Programming Gems 2*. Charles River Media, pp. 258–264.

DAWSON, B. 2002. Game Scripting in Python. In *Proceedings of the 2002 Game Developers Conference*.

DECHTER, R. AND PEARL, J. 1985. Generalised Best-First Search Strategies and the Optimality of A*. *Journal of the ACM 32*(3), pp. 505–536.

DELOACH, S. 1999. Multiagent Systems Engineering: A Methodology And Language for Designing Agent Systems. In *Proceedings of Agent-Oriented Information Systems (AOIS) '99*, pp. 45–57.

DIJKSTRA, E. W. 1959. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik 1*, pp. 269–271.

DOYLE, P. 1999. Virtual Intelligence from Artificial Reality: Building Stupid Agents in Smart Environments. In *AAAI '99 Spring Symposium on Artificial Intelligence and Computer Games*.

DOYLE, P. 2002. Believability through Context. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '02)*, pp. 342–349.

DOYLE, P. O. 2004. *Annotated Worlds for Animate Characters.* PhD thesis, Stanford University.

DYBSAND, E. 2003. AI Middleware: Getting into Character. *Game Developer 10*(8), pp. 6–10.

DYBSAND, E. 2004. Goal-Directed Behaviour Using Composite Tasks. In *AI Game Programming Wisdom 2.* Charles River Media, pp. 237–245.

ERRA, U., DE CHIARA, R., SCARANO, V. AND TATAFIORE, M. 2004. Massive Simulation Using GPU of a Distributed Behavioral Model of a Flock with Obstacle Avoidance. In *Proceedings of Vision, Modeling and Visualization 2004 (VMV).*

EVANS, R. 2001. AI in Computer Games: The Use of AI Techniques in Black & White. Seminar Notes, available from: http://www.dcs.qmul.ac.uk/research/logic/seminars/abstract/EvansR01.html. [Accessed 29/02/2008].

FAIRCLOUGH, C., FAGAN, M., MAC NAMEE, B. AND CUNNINGHAM, P. 2001. Research Directions for AI in Computer Games. Tech. Rep. TCD-CS-2001-29, Trinity College, Dublin.

FALISE, S. 2000. *Bots Behaving Badly (Making bots behave more human).* Master's thesis, Utrecht School of the Arts.

FLEMING, J. 2007. Down the Hyper-Spatial Tube: Spacewar and the Birth of Digital Game Culture. Available from: http://www.gamasutra.com. [Accessed 29/02/2008].

FORBUS, K. D. AND HINRICHS, T. R. 2006. Companion Cognitive Systems: A step towards human-level AI. *AI Magazine 27*(2), pp. 83–95.

FORBUS, K. D. AND WRIGHT, W. 2001. Some notes on programming objects in The Sims™. Class Notes, available from: http://qrg.northwestern.edu/papers/papers.html. [Accessed 29/02/2008].

FU, D. AND HOULETTE, R. 2002. Putting AI in Entertainment: An AI Authoring Tool for Simulation and Games. *IEEE Intelligent Systems 17*(4), pp. 81–84.

FU, D. AND HOULETTE, R. 2004. The Ultimate Guide to FSMs in Games. In *AI Game Programming Wisdom 2*. Charles River Media, pp. 283–302.

FU, D., HOULETTE, R. AND JENSEN, R. 2003. A Visual Environment for Rapid Behavior Definition. In *BRIMS 2003: Proceedings of the Twelfth Conference on Behavior Representations in Modeling and Simulation.*

FUNGE, J. D. 1998. *Making Them Behave: Cognitive Models for Computer Animation.* PhD thesis, University of Toronto.

FUNGE, J. D. 1999. *AI for Games and Animation: A Cognitive Modeling Approach.* A K Peters.

GARCÉS, D. 2006. Scripting Language Survey. In *Game Programming Gems 6.* Charles River Media, pp. 323–340.

GARLAN, D. AND SHAW, M. 1994. An Introduction to Software Architecture. Tech. Rep. CMU/SEI-94-TR-21, ESC-TR-94-21, Carnegie Mellon University, Pittsburgh, PA. CMU Software Engineering Institute.

GELFOND, M. AND LIFSCHITZ, V. 1998. Action Languages. *Linköping Electronic Articles in Computer and Information Science 3*(16).

GILL, S. 2004. Visual Finite State Machine AI Systems. Available from: http://www.gamasutra.com. [Accessed 29/02/2008].

GIUNCHIGLIA, E. AND LIFSCHITZ, V. 1999. Action Languages, Temporal Action Logics and the Situation Calculus. *Linköping Electronic Articles in Computer and Information Science 4*(40).

GIVEN, D. 2002. The inComplete SCUMM Reference Guide. Available from: http://www.scummvm.org. [Accessed 29/02/2008].

GLASSER, J. A. AND SOH, L. 2004. AI in Computer Games: From the Player's Goal to AI's Role. Tech. rep., University of Nebraska, Lincoln.

GOULD, D. 2002. *Complete Maya Programming: An Extensive Guide to MEL and the C++ API.* Morgan Kaufmann.

GRAEPEL, T., HERBRICH, R. AND GOLD, J. 2004. Learning to Fight. In *Proceedings of CGAIDE 2004 5th Game-On International Conference on Computer Games: Artificial Intelligence, Design and Education*, pp. 193–200.

HÄHNEL, D., BURGARD, W. AND LAKEMEYER, G. 1998. GOLEX - Bridging the Gap between Logic (GOLOG) and a Real Robot. In *KI '98: Proceedings of the 22nd Annual German Conference on Artificial Intelligence*, vol. 1505 of *LNAI*, pp. 165–176.

HARMON, M. 2004. A System for Managing Game Entities. In *Game Programming Gems 4*. Charles River Media, pp. 69–83.

HARMON, M. 2005. Building Lua into Games. In *Game Programming Gems 5*. Charles River Media, pp. 115–128.

HAYWARD, D. 2007. Uncanny AI: Artificial Intelligence in the Uncanny Valley. Available from: http://www.gamasutra.com. [Accessed 29/02/2008].

HEGDE, M. 2005. Physics, Gameplay and the Physics Processing Unit. White Paper, available from http://www.ageia.com/. [Accessed 29/02/2008].

HERRIOT, R. G. 1977. Towards the Ideal Programming Language. In *Proceedings of the ACM conference on language design for reliable software 1977*, pp. 56–62.

HIGGINS, D. 2002a. Generic A* Pathfinding. In *AI Game Programming Wisdom*. Charles River Media, pp. 114–121.

HIGGINS, D. 2002b. Terrain Analysis in an RTS - The Hidden Giant. In *Game Programming Gems 3*. Charles River Media, pp. 268–284.

HORSWILL, I. D. 2000. Functional Programming of Behaviour-Based Systems. *Autonomous Robots 9*(1), pp. 83–93.

HOULETTE, R., FU, D. AND ROSS, D. 2001. Towards an AI Behavior Toolkit for Games. In *Proceedings of AAAI Spring Symposium Series on AI and Interactive Entertainment 2001*, pp. 50–53.

HOULETTE, R., FU, D. AND JENSEN, R. 2003. Creating an AI Modeling Application for Designers and Developers. In *Proceedings of AeroSense-2003*, vol. 5091 of *SPIE*, pp. 164–171.

HUEBNER, R. 1997. Adding Languages to Game Engines. *Game Developer 4*(9).

HUGET, M.-P. 2002. Desiderata for Agent Oriented Programming Languages. Tech. Rep. ULCS-02-009, University of Liverpool.

IERUSALEMSCHY, R., DE FIGUEIREDO, L. H. AND CELES, W. 1996. Lua - an Extensible Extension Language. *Software: Practice & Experience 26*(6), pp. 635–652.

IERUSALEMSCHY, R., DE FIGUEIREDO, L. H. AND CELES, W. 2005. The Implementation of Lua 5.0. *Journal of Universal Computer Science 11*(7), pp. 1159–1176.

IERUSALEMSCHY, R., DE FIGUEIREDO, L. H. AND CELES, W. 2007. The Evolution of Lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 2–1–2–26.

JACOBS, S. 2005. Visual Design of State Machines. In *Game Programming Gems 5*. Charles River Media, pp. 169–175.

JOHNSON, D. AND WILES, J. 2001. Computer Games with Intelligence. *Australian Journal of Intelligent Information Processing Systems 7*, pp. 61–68.

KANE, B. 2007. SIGGRAPH: EA's Entis on Derailing the 'Commoditization Treadmill'. Gamasutra Industry News, available from: http://www.gamasutra.com. [Accessed 29/02/2008].

KELLEHER, C. 2006. Alice: Using 3D Gaming Technology to Draw Students into Computer Science. In *Proceedings of the Fourth Game Design and Technology Workshop and Conference (GDTW 2006)*, pp. 16–20.

KERNIGHAN, B. W. AND PIKE, R. 1999. Using Macros to Generate Code. In *The Practice of Programming.* Addison-Wesley, ch. 9.6.

KERNINGHAN, B. W. AND RITCHIE, D. M. 1988. *The C Programming Language*, second ed. Prentice Hall.

KERNINGHAN, B. W. AND VAN WYK, C. J. 1998. Timing Trials, or the Trials of Timing: Experiments with Scripting and User-Interface Languages. *Software: Practice & Experience 28*(8), pp. 819–843.

KHARKAR, S. 2004. A Modular Camera Architecture for Intelligent Control. In *AI Game Programming Wisdom 2.* Charles River Media, pp. 549–554.

KHOO, A. AND ZUBEK, R. 2002. Applying Inexpensive AI Techniques to Computer Games. *IEEE Intelligent Systems 17*(4), pp. 48–53.

KHOO, A., DUNHAM, G., TRIENENS, N. AND SOOD, S. 2002. Efficient, Realistic NPC Control Systems using Behavior-Based Techniques. In *Proceedings of AAAI Spring Symposium Series on AI and Interactive Entertainment 2002.*

KING, G. W., ATKIN, M. S. AND WESTBROOK, D. L. 2002. Tapir: the Evolution of an Agent Control Language. In *Computers and Games 2002.*

KORN, D. G. 1994. ksh - An Extensible High Level Language. In *Very High Level Languages Symposium (VHLL)*, pp. 129–146.

KORNRUMPF, A. 2005. Warum "Die Sims" sämtliche Rekorde der Spieleindustrie brach. In *Proccedings of zfxCON05 2nd Conference on Game Development*, pp. 38–45.

KOZA, J. R. 1992. *Genetic Programming: on the Programming of Computers by Means of Natural Selection.* MIT Press.

KOZA, J. R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.

KRUZEWSKI, P. 2006. Real-Time Crowd Simulation Using AI.implant. In *AI Game Programming Wisdom 3.* Charles River Media, pp. 233–248.

KUSHNER, D. 2002. The Mod Squad. *Popular Science 260*(8).

LAIRD, J. E. AND DUCHI, J. C. 2001. Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot. In *Proceedings of AAAI Spring Symposium Series on AI and Interactive Entertainment 2001*, pp. 54–58.

LAIRD, J. E. AND VAN LENT, M. 2000. Human-level AI's Killer Application: Interactive Computer Games. In *Proceedings of the 17th National Conference on Artificial Intelligence*, pp. 1171–1178.

LAIRD, J. E. AND VAN LENT, M. 2001. The Role of AI in Computer Game Genres. [book chapter] http://ai.eecs.umich.edu/people/laird/papers/book-chapter.htm.

LAIRD, J. E. 2001. It Knows What You Are Going To Do: Adding Anticipation to a Quakebot. In *Proceedings of the Agents-2001 International Conference on Autonomous Agents*, pp. 385–392.

LEE, K. H., CHOI, M. G. AND LEE, J. 2006. Motion Patches: Building Blocks for Virtual Environments Annotated with Motion Data. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pp. 898–906.

LEVESQUE, H. J., REITER, R., LESPERANCE, Y., LIN, F. AND SCHERL, R. B. 1997. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming 31*(1–3), pp. 59–83.

LEVESQUE, H., PIRRI, F. AND REITER, R. 1998. Foundations for the Situation Calculus. *Linköping Electronic Articles in Computer and Information Science 3*(18).

LI, S. 2002. Rock 'em, sock 'em Robocode! IBM developerWorks: Java technology – http://www-106.ibm.com/developerworks/library/j-robocode/. [Accessed 29/02/2008].

LIFSCHITZ, V. 1997. Two Components of An Action Language. *Annals of Mathematics and Artificial Intelligence 21*(2), pp. 305–320.

LINDHOLM, E., KILGARD, M. J. AND MORETON, H. 2001. A User-Programmable Vertex Engine. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 149–158.

LINK, W. 1995. *Assembler-Programmierung.* Franzis.

LOUI, R. P. 1996. Why GAWK for AI. *ACM SIGPLAN Notices 31*(8), pp. 8–9.

LUKE, S., HOHN, C., FARRIS, J., JACKSON, G. AND HENDLER, J. A. 1998. Co-Evolving Soccer Softbot Team Coordination with Genetic Programming. In *RoboCup-97: Robot Soccer World Cup I*, vol. 1395 of *LNCS*, pp. 398–411.

LUKE, S. 1998. Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup97. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pp. 214–222.

MACDORMAN, K. F. 2005. Androids as an Eperimental Apparatus: Why is there an Uncanny Valley and can we Exploit it? In *Proceedings of CogSci-2005 Workshop: Toward Social Mechanisms of Android Science*, pp. 106–118.

MACEDONIA, M. 2000. Using Technology and Innovation to Simulate Daily Life. *IEEE Computer 33*(4), pp. 110–112.

MAGERKO, B. 2006. Intelligent Story Direction in the Interactive Drama Architecture. In *AI Game Programming Wisdom 3.* Charles River Media, pp. 583–596.

MARK, W. R., GLANVILLE, R. S., AKELEY, K. AND KILGARD, M. J. 2003. Cg: a System for Programming Graphics Hardware in a C-like Language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pp. 896–907.

MATTHEWS, J. 2002. Basic A* Pathfinding Made Simple. In *AI Game Programming Wisdom.* Charles River Media, pp. 105–113.

MCCARTHY, J. 1955. A Proposal for the Summer Research Project on Artificial Intelligence. Available from: http://www-formal.stanford.edu/jmc/history/. [Accessed 29/02/2008].

McCarthy, J. 1959. Programs with Common Sense. In *Mechanisation of Thought Processes, Proceedings of the Symposium of the National Physics Laboratory*, pp. 77–84.

McCarthy, J. 2007. What is Artificial Intelligence. Available from: http://www-formal.stanford.edu/jmc/whatisai/whatisai.html. [Accessed 29/02/2008].

McCusky, M. 2000. Fuzzy Logic for Video Games. In *Game Programming Gems*. Charles River Media, pp. 319–329.

McIver, L. and Conway, D. 1996. Seven Deadly Sins of Introductory Programming Language Design. In *Proceedings of Software Engineering: Education and Practice (SE:E&P'96)*, pp. 309–316.

McNaughton, M., Redford, J., Schaeffer, J. and Szafron, D. 2003. Pattern-Based AI Scripting using ScriptEase. In *Proceedings of the 16th Canadian Conference on Artificial Intelligence (AI 2003)*, pp. 35–49.

Mogilefsky, B. 1999. Lua in Grim Fandango. Available from: http://www.grimfandango.net. [Accessed 29/02/2008].

Montana, D. J. 1995. Strongly Typed Genetic Programming. *Evolutionary Computation 3*(2), pp. 199–230.

Nareyek, A., Karlsson, B. F. F., Wilson, I., Chady, M., Mesdaghi, S., Axelrod, R., Porcino, N., Combs, N., El Rhalibi, A., Wetzel, B. and Orkin, J. 2004. The 2004 Report of the IGDA's Artificial Intelligence Interface Standards Committee. Available from: http://www.igda.org/ai/. [Accessed 29/02/2008].

Nareyek, A., Combs, N., Karlsson, B. F. F., Mesdaghi, S. and Wilson, I. 2005. The 2005 Report of the IGDA's Artificial Intelligence Interface Standards Committee. Available from: http://www.igda.org/ai/. [Accessed 29/02/2008].

NAREYEK, A. 2000. Intelligent Agents for Computer Games. In *Proceedings of the Second International Conference on Computers and Games (CG2000)*, pp. 414–422.

NAREYEK, A. 2007. Game AI is Dead, Long Live Game AI. *IEEE Intelligent Systems 22*(1), pp. 9–11.

OLSEN, J. R. 1991. *The Visionary Programmer's Handbook or Quilling the Great Adventure.* Oxxi.

ORKIN, J. 2002. 12 Tips from the Trenches. In *AI Game Programming Wisdom.* Charles River Media, pp. 29–35.

ORKIN, J. 2004a. Applying Goal-Oriented Action Planning to Games. In *AI Game Programming Wisdom 2.* Charles River Media, pp. 217–228.

ORKIN, J. 2004b. Symbolic Representation of Game World State: Toward Real-Time Planning in Games. In *AAAI-04 Workshop on Challenges in Game AI*, pp. 26–30.

ORKIN, J. 2006. Three States and a Plan: The A.I. of F.E.A.R. In *Proceedings of the 2006 Game Developers Conference.*

OUP. 2002. Scripting Language. *A Dictionary of Computing.* Oxford University Press.

OUSTERHOUT, J. K. 1998. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer 31*(3), pp. 23–30.

PATTIS, R. E. 1981. *Karel the Robot, a Gentle Introduction to the Art of Programming.* John Wiley and Sons.

PEMBERTON, S. AND DANIELS, M. 1982. *Pascal Implementation: The P4 Compiler and Interpreter.* Ellis Horwood.

PETERS, C., DOBBYN, S., MAC NAMEE, B. AND O'SULLIVAN, C. 2003. Smart Objects for Attentive Agents. In *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision.*

POIKER, F. 2002. Creating Scripting Languages for Nonprogrammers. In *AI Game Programming Wisdom*. Charles River Media, pp. 520–529.

PRECHELT, L. 2003. Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java. *Advances in Computers 57*, pp. 205–270.

PRINZ, P. AND CRAWFORD, T. 2006. *C in a Nutshell*. O'Reilly.

RABIN, S. 2000a. A* Aesthetic Optimizations. In *Game Programming Gems*. Charles River Media, pp. 264–271.

RABIN, S. 2000b. Designing a General Robust AI Engine. In *Game Programming Gems*. Charles River Media, pp. 221–236.

RABIN, S. 2000c. The Magic of Data-Driven Design. In *Game Programming Gems*. Charles River Media, pp. 3–7.

RABIN, S. 2002a. Finding Redeeming Value in C-Style Macros. In *Game Programming Gems 3*. Charles River Media, pp. 26–37.

RABIN, S. 2002b. Implementing a State Machine Language. In *AI Game Programming Wisdom*. Charles River Media, pp. 314–320.

RABIN, S. 2004. Promising Game AI Techniques. In *AI Game Programming Wisdom 2*. Charles River Media, pp. 15–27.

RELIC ENTERTAINMENT. 2003. *SCAR - Scripting at Relic*.

REYNOLDS, C. W. 1987. Flocks, Herds and Schools: A Distributed Behavioral Model. *Computer Graphics 21*(4), pp. 25–34.

REYNOLDS, C. W. 1994. Competition, Coevolution and the Game of Tag. In *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pp. 59–69.

REYNOLDS, C. W. 1999. Steering Behaviors for Autonomous Agents. In *Proceedings of the 1999 Game Developers Conference*.

REYNOLDS, C. W. 2006. Big Fat Crowds on PS3. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames.*

RITTER, F. E., HAYNES, S. R., COHEN, M., HOWES, A., JOHN, B., BEST, B., LEBIERE, C., JONES, R. M., CROSSMAN, J., LEWIS, R. L., ST. AMANT, R., MCBRIDE, S. P., URBAS, L., LEUCHTER, S. AND VERA, A. 2006. High-level Behavior Representation Languages Revisited. In *Proceedings of ICCM - 2006- Seventh International Conference on Cognitive Modeling*, pp. 404–407.

ROBERTS, M. B. V. 1971. *Biology: A Functional Approach.* Nelson.

RUSSEL, S. J. AND NORVIG, P. 1995. *Artificial Intelligence: A Modern Approach.* Prentice Hall.

SARAFOPOULOS, A. 2001. GP Interface. [function library]. NCCA, Bournemouth University.

SCHAFFER, E. AND WOLF, M. 1991. The UNIX Shell as a Fourth Generation Language. Tech. rep., Revolutionary Software. Available from: http://www.rdb.com.

SCHNEIDER, J. G. AND NIERSTRASZ, O. 1999. Components, Scripts and Glue. In *Software Architectures - Advances and Applications.* Springer-Verlag, pp. 13–25.

SCHWARTZ, R. L. 1992. *Learning Perl.* O'Reilly.

SCOTT, B. 2002a. Architecting an RTS AI. In *AI Game Programming Wisdom.* Charles River Media, pp. 397–401.

SCOTT, B. 2002b. The Illusion of Intelligence. In *AI Game Programming Wisdom.* Charles River Media, pp. 16–20.

SEARLE, J. R. 1980. Minds, Brains, and Programs. *Behavioral and Brain Sciences 3*(3), pp. 417–457.

SHAY, X. 2004. Polymorphism in Angelscript. Available from: http://www.gamedev.net. [Accessed 29/02/2008].

SHERROD, A. 2007. *Ultimate 3D Game Engine Design & Architecture.* Charles River Media.

SHI, Y., GREGG, D., BEATTY, A. AND ERTL, M. A. 2005. Virtual Machine Showdown: Stack versus Registers. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pp. 153–163.

SIEGEL, E. V. AND CHAFFEE, A. D. 1996. Genetically Optimizing the Speed of Programs Evolved to Play Tetris. In *Advances in Genetic Programming, Volume 2.* MIT Press, pp. 279–298.

SIEM, K. V. 2006. Artificial Intelligence in Computer Games. Project Report, available from http://www.unl.csi.cuny.edu/~siem/. [Accessed 29/02/2008].

SIMPSON, C. 2002. Past, Present and Future of AvP2 Modifications. Available from http://www.planetavp.com/features/articles/sub-editorial-13.shtml. [Accessed 29/02/2008].

SKIBAK, S. AND STAHL, M. 2002. KI - State of the Art. Available from http://www.uni-ulm.de/~s_hdamme/. [Accessed 05/04/2004].

SNAVELY, P. J. 2004. Empowering Designers: Defining Fuzzy Logic Behaviour through Excel-Based Spreadsheets. In *AI Game Programming Wisdom 2.* Charles River Media, pp. 541–548.

SNAVELY, P. J. 2006. Custom Tool Design for Game AI. In *AI Game Programming Wisdom 3.* Charles River Media, pp. 3–12.

SNOOK, G. 2000. Simplified 3D Movement and Pathfinding Using Navigation Meshes. In *Game Programming Gems.* Charles River Media, pp. 288–304.

SPIRIG, M., ZERBST, S., ANDERSON, E. F., PECH, S., ENGEL, S. AND DÜVEL, O. 2003. ZFX Bot Contest. [Private on-line discussion 18/03/2003–19/05/2003].

STEELE, G. L. AND GABRIEL, R. P. 1993. The Evolution of Lisp. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pp. 231–270.

STOUT, B. 2000. The Basics of A* for Path Planning. In *Game Programming Gems*. Charles River Media, pp. 254–263.

STROUSTRUP, B. 1991. What is "Object-Oriented Programming"? In *Proceedings of the 1st European Software Festival.*

STROUSTRUP, B. 1997. *The C++ Programming Language*, third ed. Addison Wesley.

STROUSTRUP, B. 2005. The Design of C++0x. *C/C++ Users Journal 23*(5).

SWEETSER, P. AND WILES, J. 2005. Scripting versus Emergence: Issues for Game Developers and Players in Game Environment Design. *International Journal of Intelligent Games and Simulations 4*(1), pp. 1–9.

SWEETSER, P. 2003. Current AI in Games: A Review. Tech. rep., University of Queensland, Brisbane.

TANENBAUM, A. S. 2001. *Modern Operating Systems*, second ed. Prentice Hall.

TAPPER, P. 2003. Personality Parameters: Flexibly and Extensibly Providing a Variety of AI Opponents' Behaviors. Available from: http://www.gamasutra.com. [Accessed 29/02/2008].

TOMLINSON, S. L. 2004. The Long and Short of Steering in Computer Games. *International Journal of Simulation: Systems, Science & Technology 5*(1–2).

TOZOUR, P. 2001. Influence Mapping. In *Game Programming Gems 2.* Charles River Media, pp. 287–297.

TOZOUR, P. 2002a. Building a Near-Optimal Navigation Mesh. In *AI Game Programming Wisdom.* Charles River Media, pp. 171–185.

TOZOUR, P. 2002b. The Evolution of Game AI. In *AI Game Programming Wisdom.* Charles River Media, pp. 3–15.

TOZOUR, P. 2002c. The Perils of AI Scripting. In *AI Game Programming Wisdom*. Charles River Media, pp. 541–547.

TURING, A. M. 1950. Computing Machinery and Intelligence. *Mind 59*.

UNTCH, R. H. 1990. Teaching Programming Using the Karel the Robot Paradigm Realized with a Conventional Language. Available from: http://www.mtsu.edu/~untch/karel/karel90.pdf. [Accessed 29/02/2008].

VAN DEURSEN, A., KLINT, P. AND VISSER, J. 2000. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices 35*(6), pp. 26–36.

VAN LENT, M. AND LAIRD, J. E. 1999. Developing an Artificial Intelligence Engine. In *Proceedings of the 1999 Game Developers Conference.*

VAN LENT, M., LAIRD, J., BUCKMAN, J., HARTFORD, J., HOUCHARD, S., STEINKRAUS, K. AND TEDRAKE, R. 1999. Intelligent Agents in Computer Games. In *Proceedings of the 16th National Conference on Artificial Intelligence*, pp. 929–930.

VARANESE, A. 2003. *Game Scripting Mastery.* Premier Press.

WALLIS, A. 2007. Is Modding Useful? In *Game Carreer Guide 2007.* CMP Media, pp. 25–28.

WARREN, P. 2001. Teaching Programming Using Scripting Languages. *Journal of Computing Sciences in Colleges 17*(2), pp. 205–216.

WELSH, S. AND PISAN, Y. 2005. Information-Oriented Design and Game AI. In *Proceedings of the Second Australasian Conference on Interactive Entertainment*, pp. 227–234.

WEST, M. 2007. Domain-Specific Languages. *Game Developer 14*(7), pp. 33–36.

WILCOX, B. 2007. Reflections on Building Three Scripting Languages. Available from: http://www.gamasutra.com. [Accessed 29/02/2008].

WILSON, K. 2002. Data-Driven Design. Blog entry, available from: http://www.GameArchitect.net, May. [Accessed 29/02/2008].

WIRTH, N. 1973. The Programming Language Pascal (Revised Report). Tech. Rep. 5, Eidgenössische Technische Hochschule Zürich. Institut für Informatik.

WIRTH, N. 1986. *Compilerbau.* Teubner.

WIRTH, N. 1993. Recollections about the Development of Pascal. *ACM SIGPLAN Notices 28*(3), pp. 333–342.

WIRTH, N. 1996. *Compiler Construction.* Addison-Wesley.

WIRTH, N. 2006. Good Ideas, through the Looking Glass. *IEEE Computer 39*(1), pp. 28–39.

WOODCOCK, S. 2001. AI Roundtable Report. In *Proceedings of the 2001 Game Developers Conference.*

WRIGHT, I. AND MARSHALL, J. 2000. More AI in less Processor Time: 'Egocentric' AI. Available from: http://www.gamasutra.com. [Accessed 29/02/2008].

YOB, G. 1975. Hunt the Wumpus. *Creative Computing.*

YUE, B. AND DE BYL, P. 2006. The State of the Art in Game AI Standardisation. In *Proceedings of the 2006 international Conference on Game Research and Development*, pp. 41–46.

ZERBST, S., DÜVEL, O. AND ANDERSON, E. 2003. *3D-Spieleprogrammierung.* Markt + Technik.