Modelling and Transformation of Artificial Intelligence for Non-Player Characters

Gino Wuytjens

Supervisor: Prof. Dr. Hans Vangheluwe

Faculty of Science
University of Antwerp
Antwerp, Belgium

May 1st, 2012

A thesis submitted to University of Antwerp in
fulfilment of the requirements of the degree of
Master of Science in Information Technology

Abstract

Artificial Intelligence for virtual characters is currently developed through a textual interface. <Model Driven Development and Statecharts are a natural fit> <Statecharts provide lots of advantages> <Model Transformation> <Framework Design> <Python Techdemo> <Commercial product> <Possibilities going forward>

Contents

1 Introduction

This introductory chapter serves to provide the reader with the required context to understand the problem faced by this Master's Thesisand the objectives that were posed at the start of the academic year. Furthermore, a summary of various considerations that had to be made before work on the thesis could be started is provided, such as the choice of programming language, modelling formalism and modelling tools. The chapter ends with some insight into my personal motivation to work on this particular thesis and the outline of the thesis as a whole.

1.1 Problem Statement

In previous work, it has been shown that modelling Artificial Intelligence of game characters and robots is possible. In [KDV07], a layered approach is formulated, which is expanded upon in [Bla08] to show that the Statecharts formalism is a natural fit for this purpose and that the developed AI can be easily transformed to use in various application domains, from a computer game to a real-life robot.

Non-Player Characters in virtual environments or computer games show behaviour powered by some form of AI. A very popular and widespread technique is to have the NPCs be *stateful*. For example, an AI controlled tank in a Tank Wars type of game might have three possible states: *idle*, *attacking* and *fleeing*. The actual state of the tank will determine the actions the tank can take. This technique has been in use for a long time and has shown itself to be exceptionally useful to quickly develop interesting AI for games. However, the technique has not really evolved over the years. While computer graphics and gameplay have gone through big changes the last several decades, AI development for NPCs has not seen the same kind of change. This thesis proposes to leverage the findings of [KDV07] and [Bla08] to model the AI of NPCs completely in the Statecharts formalism, making use of all the features provided by the language to create a new paradigm for the creation of responsive behaviour in NPCs.

Computer games that involve any kind of Non-Player Charactermake use of AI to create realistic or otherwise enjoying experiences. Particularly in Masively Multiplayer Online Games, players often encounter large groups of the same NPCs, usually powered by the exact same AI, but with differing parameters. This causes players to lose their immersion in the interactive experience and start seeing it as a series of encounters that must be completed to continue the experience. It should be possible then, to create some AI for a particular type of NPC and have this AI transformed into several variants. Such variants would cause groups of NPCs to show differing behaviour, which differs on more than simply the amount of health, speed or other simple attributes. This heterogenity in NPC groups of the same type can be used to increase player immersion and simulation realism. It could potentially be used to tailor the game experience to the player, by transforming the original AI according to parameters retrieved from the player's performance in previous parts of the game.

1.2 Objectives

The problem statement above highlights the current situation in AI development for NPCs and shows the areas which can be improved. To achieve the proposed functionality, several objectives were formulated that need to be fulfilled in order to achieve our goal.

Research Related Work

First and foremost, related work in the fields of AI modelling, model transformation and NPC AI practices should be explored. Being aware of the current state of these fields is important to ensure the validity of this thesis, since it will build upon the previous work to tackle the previously mentioned problems.

Test Platform

As the first actual implementation task, a test platform should be developed that can be used to test the developed AI in a controlled environment. The platform should mirror the architecture of current commercial games closely, in order to face the same difficulties as we would in a commercial product, but also allowing us create AI that can later be used in such a commercial product.

AI Framework

A framework should be developed to facilitate the modelling of AI and the compilation of the AI to the required programming language. The framework should be as seperate from the test platform in order to allow it to be used for various destination platforms.

AI Transformations

Once it is possible to model AI, compile it and subsequently run the generated code on our test platform, transformation rules should be formulated to allow re-use of a single AI model, with properties as described in the problem statement above. The AI framework should be expanded to ease the execution of transformations on previously constructed models.

Analysis Platform

Now that AI can be modelled and transformed, we should test how the transformations affect NPC behaviour. Since it would be less than desireable to look at the behaviour of a large group of AI variants manually, the test platform should be expanded to allow automated testing and analysis of the variants. Analysis should be performed through one or more interchangeable heuristics; at one point we might be looking for AI variants that are good at playing the game, but another time we might be interested in finding AI variants that provide a good game experience when pitted against a human player, which is not neccesarily the same thing.

Commercial Product

The AI framework developed over the course of this thesis should be expanded to allow us to test modelling of AI in an actual commercial product. This would show that this approach is not just academic in nature, but that it can actually be applied to existing products. The game chosen should be popular and advanced enough such that real-world application possiblities are presented to a wide audience.

1.3 Considerations

This thesis builds upon previous work in the fiels of AI modelling, transformation and current NPC AI practices. Because of this, the decisions that had to be made at the start of this thesis are heavily influenced by this body of previous work. Chapter 4 provides an overview of previous work, providing the reader with the required context.

Modelling Formalism

Due to the previous work in [KDV07] and [Bla08], the choice of modelling formalism was obvious. This thesis will build upon the techniques discussed in these papers, modelling AI using the *Statecharts* formalism.

Modelling Environment

Due to my association with the Modelling, Simulation and Design lab headed by my supervisor, Prof. Dr. Hans Vangheluwe, I have chosen to use the *AToM*^{3} tool to model all statecharts used throughout this thesis. The tool provides all functionality required to create models in the *Statecharts* formalism and perform transformations on these models. Furthermore, several other tools, such as the compiler used to compile statechart models into the required programming language (*SCC*), are integrated into *AToM*^{3}, which eases the workflow proposed by the framework that will be developed throughout this thesis.

Programming Language

Finally, I had to choose a programming language in which I would develop a test platform. There were various choices such as *Java*, *C++* and *Python*. Since I already had quite some experience with Java and C++, I chose for the language I was least familiar with, namely Python. The language furthermore allows a developer to rapidly create a prototype application and test out designs and functionalities. Also, the third party *PyGame* library provides packages to facilitate the development of games, mirroring the architecture used in commercial games today.

1.4 Personal Motivation

Before starting my education at the University of Antwerp, I was very much intrigued by computer games and the ways in which they allow a story to be told interactively. This sparked my interest in the field of computer science and led me to start my current education. I have always felt computer games were not utilised to their full potential as a medium to convey experiences and stories. Because of their interactive nature, stories can become dynamic, changing in accordance with decisions made by a player. However, the same kind of dynamic nature cannot be found in individual scenes or 'encounters' in games. Encounters with friendly villagers or ferocious beasts quickly become stale and uninteresting when such NPCs always show exact same behaviour. For example, in the real world, a deer hunter might have to adjust his hunting strategy for each individual animal he is hunting, because some deer are more shy than others or have varying types of reactions to an unknown smell in their vicinity. This type of variation is thus far unavailable in computer games, causing players to quickly deduce the behaviour patterns for the various NPC types they encounter, rendering the experience boring and stale. I feel that through AI modelling and transformation, NPC AI can be varied to emulate the varying behaviour of individuals in the real world. This is but one application of the technologies discussed in the coming chapters, but it is one which I was thinking about long before starting my work on this thesis.

Of course, apart from the opportunity to develop something new which the world has never before seen, I also have more selfish motivations. I feel this thesis is an excellent opportunity to work on a big project from start to finish, learning in detail how each stage of software development and research is performed. Previous courses at the university usually let us work on projects for a maximum of two months, much less time than real world software projects. I also look forward to exploring the topics of Artificial Intelligence, Modelling and Model Transformation in detail. I only got in touch with modelling in my second year at university, but it immediately caught my attention. I was thus far unaware of this new paradigm in computer programming and exploring it has really opened my eyes to new perspectives on software development, perspectives I hope to expand even further through my work on this thesis.

1.5 Thesis Outline

Chapter 2 will begin this thesis with a general introduction of the *Statecharts*modelling formalism used throughout this thesis. Afterwards in chapter 3, we provide an introduction to model transformation. After these concepts have been explored, the thesis closely follows the structure proposed by the objectives mentioned above. Chapter 4 explores the related work in the areas of AI modelling, transformation and current NPC AI practices. This related work serves as the foundation for the new developments introcuded throughout this thesis. Chapter 5 will chronicle the development of the AI framework and chapter 6 serves the same function for the transformation submodule of the framework. Finally, chapter 7 explores the possibility of leveraging the developed framework for use in an actual commercial product. In the conclusion, we provide arguments for the real world applicability of this thesis and explore how further work in the field of AI modelling and transformation may open up even more ways to improve NPC behaviours.

2 Statecharts Formalism

Before we can model Artificial Intelligence, we first need to obtain an understanding of the modeling formalism we will use to achieve this goal. In this chapter, an introduction the syntax and semantics of the *Statecharts* formalism is provided. For detailed information regarding the concepts discussed within this chapter, please refer to the cited sources. As was mentioned in the introductory chapter, this thesis builds upon previously published work. With regards to the modeling of AI using Statecharts, the most recent research upon this thesis is based was published in [Bla08], which itself was based on work published in [KDV07]. An overview of the concepts defined in work related to this thesis can be found in chapter 4.

2.1 Introduction

David Harel introduced the *Statecharts* formalism to the world in his publication *Statecharts: A Visual Formalism For Complex Systems* [Har87]. His motivation was the fact that, at the time, no suitable method had been found to describe large and complex *reactive* systems. Harel specifically created the *Statecharts* formalism to address this problem.

Reactive systems, as opposed to transformational systems, specify an interaction between themselves and their environment. This inherently means they are event-based, continually reacting to external or internal stimuli. Some examples of reactive systems are telephones, cars, traffic control systems, and so on. Even the lights in our homes can be thought of as a reactive system, responding to button presses and the flow of electricity.

Before the introduction of Statecharts, several methods existed to specify reactive systems, but they were generally accepted to only be applicable to relatively simple problems. Traditional state diagrams, such as the one seen in figure 2.1, are inadequate for describing complex reactive systems. These diagrams are simply directional graphs. Nodes in the graph denote states, where as arrows – labeled with the triggering event and optional guarding condition – denote transitions between these states. Referring to figure 2.1, the states are A, B and C, while the transitions are the arrows between them. Furthermore, the labels α, β and γ denote events, while P denotes a certain condition.

Such state diagrams contain no notion of depth (or state hierarchy) or parallelism (or orthogonality), which would cause a very large build up in the amount of states required to accurately describe

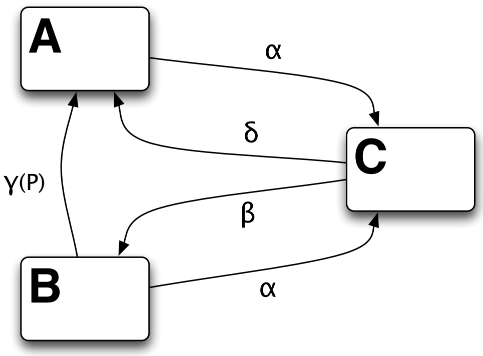

Figure 2.1: A State Diagram

a system. Furthermore, there is no way for one state to influence other states in the same system, because there is syntactic construct allowing for the broadcasting of events (in other words, the system cannot react to or even generate internal stimuli). Due to these shortcomings, D. Harel set out to create a visual formalism that would solve these issues, while at the same time allowing for visual inspection of the system. D. Harel summarizes this in [Har87] with the following 'equation':

$$\begin{aligned}\textsf{\textit{Statecharts}} = & \, State\ Diagrams \\ & + Depth \\ & + Orthogonality \\ & + Broadcast\ communication\end{aligned}$$

D. Harel only provided a brief description of how one could implement the semantics of Statecharts in his original paper. In one of his later works, *The STATEMATE Semantics of Statecharts* [HN96], he expands upon this to provide a complete formal semantics for the language. This was in fact the first attempt at formalizing the semantics of the Statecharts formalism, and it has since been adopted by various modeling packages and applications. Throughout this thesis, we will refer to the STATEMATE semantics to define the formal behavior of the models that are developed.

The *Statecharts* formalism's syntax is based upon traditional state diagrams and the concept of a higraph, which allows for the expression of state hierarchies. In what follows, we shall go over the various syntactic constructs provided by the Statecharts formalism. Keep in mind that the exact representation of states might differ between editors, some utilize rectangles where others use circles. For the most part however, these differences are limited to trivial details that have no bearing on the actual models that can be developed. We will use the syntax as proposed in [Har87] to illustrate the various concepts. As a special note, we will end this section by taking a look at how *AToM*^{3} implements the Statecharts syntax.

2.2 States

Single states are represented by labeled rectangles, as seen in figure 2.2. Here we see the model of a trivial system, consisting of only one state A. The unlabeled transition arriving at A signifies that this is the initial state of the system. Since no other transitions are present, the system cannot change states.

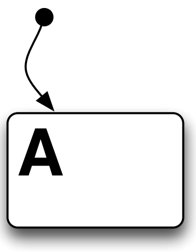

Figure 2.2: Single Statecharts state

Figure 2.3 shows a system consisting of several states, with transitions between them labeled in the form *Event(condition)/Action*. The system consists of states A, B and C, where A is the initial state of the system. State A has one outgoing transition to state C, with the label $\alpha(Q)/\delta$, which signifies this transition will be taken when an event α occurs, but only if condition Q evaluates as *true*. The action, δ signifies an event that should be broadcast when this transition is taken. This event could in turn form the label of another transition in the system. Note that all parts of the label – the event, condition and action – are optional, and no transition besides the one just discussed includes all these components. It is thus also possible for a transition to exist without any label, signifying an instantaneous transition between states.

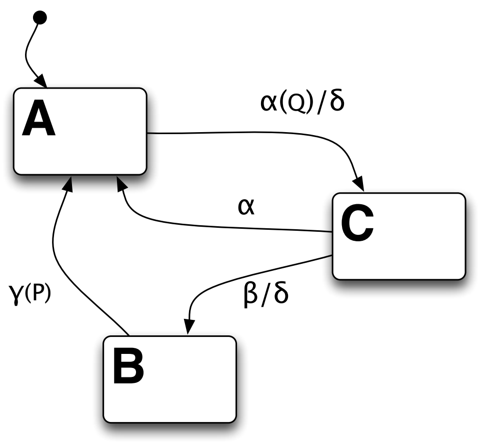

Figure 2.3: Multiple Statecharts states with labeled transitions

2.3 Depth: Compound States

With the constructs discussed above, we can already start describing simple reactive systems, but no notion of state hierarchy is yet available. We should be able to *cluster* states together into *compound states* or *refine* states into multiple *substates*. This clustering and refinement is made possible through XOR decomposition of states, meaning that when a compound state is active, exactly one of its substates must be active.

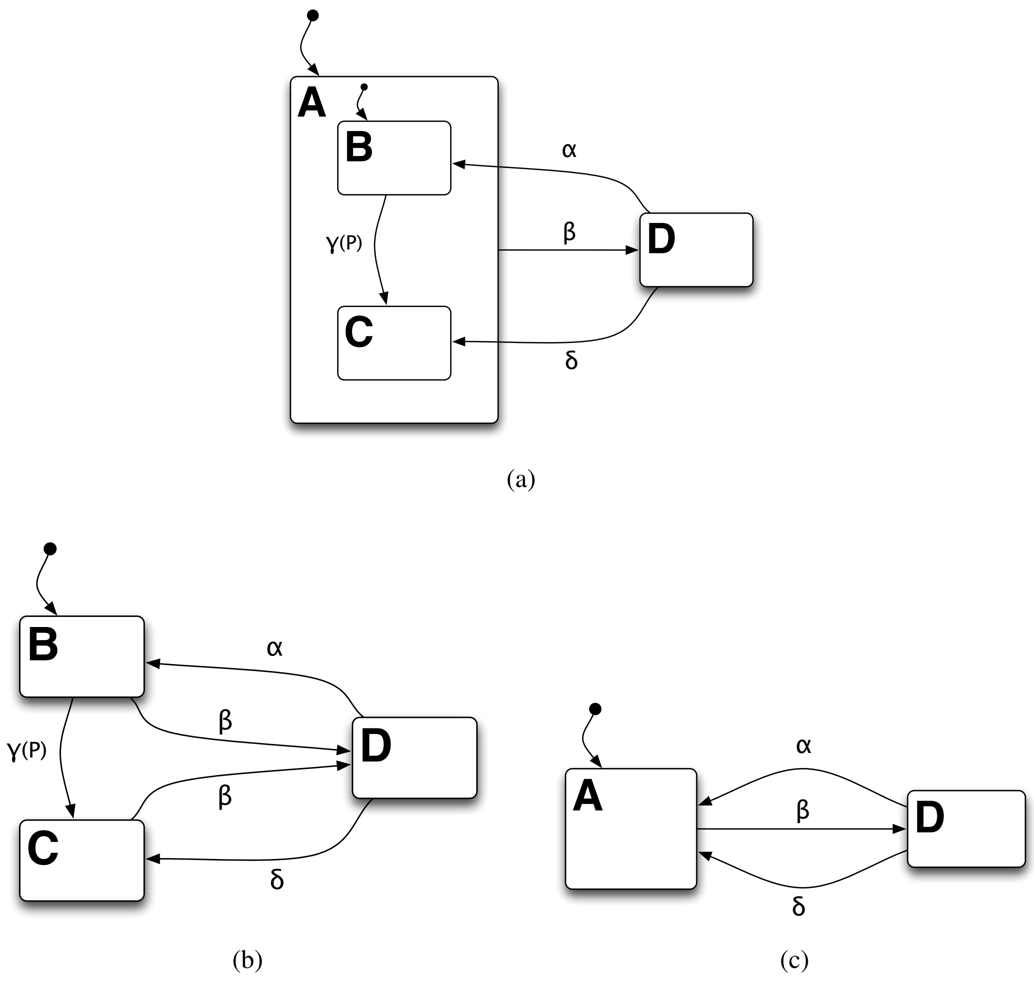

Figure 2.4: State hierarchy

Figure 2.4a shows a system consisting of four states. State A is a compound state, consisting of the substates B and C. We see that state A is the initial state of the system and B is the initial state that will become active as soon as state A becomes active. Each level of hierarchy needs to specify an initial state, or the whole system would be underdefined. We can furthermore see that transitions may depart from any level of hierarchy and arrive at any other. Figure 2.4b is the same system, this time modeled without making use of compound states, causing the need for two transitions triggered by event β instead of one departing from the compound state. In essence, we can arrive at (a) by clustering the states B and C in (b). Figure 2.4c on the other hand models a less detailed system. By refining state A from (c) into the two substates B and C, we reached (a).

2.4 Parallelism: Orthogonal States

Now that we know how to structure the system's model into state hierarchies, let's take a look at how the Statecharts formalism provides us with the ability to model parallelism, in Statecharts terms referred to as *orthogonality*. Where XOR decomposition allowed one substate of a compound state to be active at any time, orthogonality is realized through AND decomposition into *orthogonal states*. This effectively signifies that if an orthogonal state is active, all of its substates must be active too.

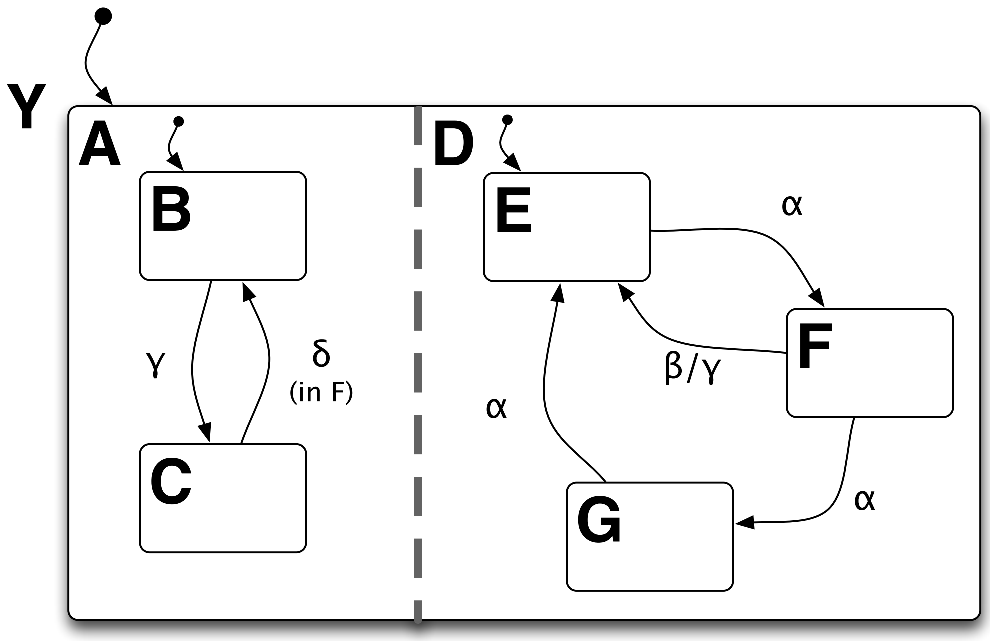

Figure 2.5: System consisting of one orthogonal state

Figure 2.5 shows a system modeled using a single orthogonal state Y, which contains two substates A and D. Since Y is the initial state of the system (at the highest level in the state hierarchy), both A and D become active as soon as the system initiates. The A and D substates of Y are themselves compound states, consisting of 2, respectively 3 substates themselves.

Orthogonal states is the mechanism through which the Statecharts formalism avoids the infamous 'state blowup' problem, where the models of complex systems would contain so many states and transitions that the complete model loses some of its most desirable properties: human readability and independent components. The system modeled in figure 2.5 could also be modeled without an orthogonal state, by flattening the model. The result of this can be seen in figure 2.6. Even for this very simple model, the flattened version is a lot less readable. If we would name the states arbitrarily (as opposed to using the names of the original states), we would be hard pressed to find the different components in the system, while the model in figure 2.5 inherently shows us how the system is organized in the two components A and D.

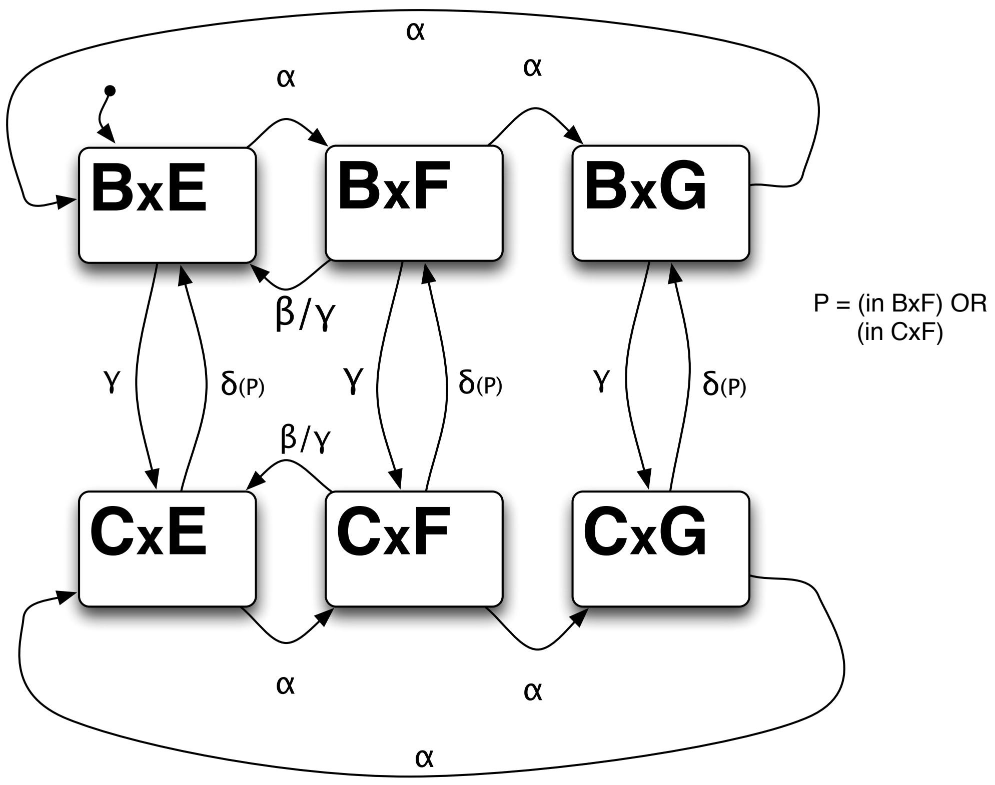

Figure 2.6: Same system, modeled without orthogonal state by flattening the original model

Apart from more obvious advantages such as readability and independent components, we can now also use special conditions in our models that check if the system is in some specified state. Refer to the model in figure 2.5, the transition from state C to B: *in F*, specifies that the transition can only be taken if state F is currently active. Note that such a condition was impossible to formulate before the introduction of orthogonal states! Furthermore, the transition from state F to state E contains an action: event γ is to be broadcast when this transition is taken. In the next step (see [HN96]) after this transition, event γ can be 'sensed' by other states and/or transitions in this system. In fact, if state B is active and the transition from state F to E is taken, in the next step the transition from B to C will be triggered by the broadcasted γ event.

2.5 Formal Semantics

Since *Statecharts* models define the behavior of a system, we must also define some way to execute this behavior. In other words, we want to simulate these models. Reliable simulation can only be performed when the syntactic constructs discussed in the previous chapters are associated with formal semantics. Furthermore, since the formalism itself contains no notion of time, we should define the concept of time as it relates to the simulation of Statecharts models.

There are two distinct methods to defining time, the first approach is discrete time, in which simulation occurs in steps, each step advancing the current time by one time unit. Formal semantics will then define what behavior of the model will be executed during each step, based on the state of the system at the start of each time step. Some of the questions that should be answered by a formal semantics are:

- Should transition actions be performed in the same step as the one in which the transition is taken, or in the next?
- In a step in which a transition is taken between source state x and destination state y: Which of the special conditions *in X* and *in Y* should evaluate as true? Maybe they should both evaluate as true, or perhaps as false?

The second approach to defining time takes the form of continuous time. Time advances normally during actions and simulation of the model, which corresponds to the natural notion of time as we know it. In this case, the formal semantics must define how the orthogonal components of the system can be simulated, such that the parallel and independent properties of Statecharts are preserved.

2.5.0.1 STATEMATE Semantics

As was previously mentioned, throughout this thesis we will refer to the formal semantics proposed in [HN96], in the following referred to as the Statemate semantics. These semantics define reliable simulation of Statecharts models in discrete time. Since the goal of this thesis is to define the behavior of entities in a virtual world in which time advances continuously, we will have to find a way to translate this continuous time into the time steps defined in the semantics.

TODO Give some example answers to questions that arise from the various syntactic constructs. Perhaps focus on how the event broadcast system is defined, since this was not discussed in the above.

2.6 Statecharts in *AToM*^{3}

TODO Create some example statecharts in atom3 to show how the various syntactic constructs are implemented in the atom3 environment. Since there are some minor differences to the more general illustrations in the above discussion, this section should ensure readers are not confused by the illustrations in later parts of the thesis.

3 Model Transformation

Model transformation consists of the creation and application of *transformation rules* which *transform* their model input to some output. This chapter will serve as an introduction to model transformation itself and how it will be applied in this thesis.

3.1 Introduction

Model transformation is a very important area of research in the field of model-driven engineering. It is a tool through which efficiency of software development can be greatly improved by automated transformation of input models. There are many forms such transformations may take and an effort to classify the various types of possible model transformations was made by K. Czarnecki and S. Helsen in *Classification of Model Transformation Approaches (2003)* [CH03]. This work clearly shows that model transformation consists of much more than simply defining transformation rules. This can be clearly seen in figure 3.1, representing the various properties of model transformations. In what follows we will go over these properties and find out how they are represented in this thesis.

3.2 Features

3.2.1 Transformation Rules

When dealing with model transformation, one creates transformation rules to describe the required transformations. These rules define how a model can be transformed into some output. This output is not necessarily another model, it could also be some textual representation of the original input. Transformation rules consist of a left-hand side and a right-hand side (further referred to as the LHS and RHS respectively). Both LHS and RHS may be formulated in terms of variables, patterns and/or logic.

Variables
: Variables representing meta-data about the (source and/or target) models may be used in the transformation rule to influence the actual transformation.

Patterns

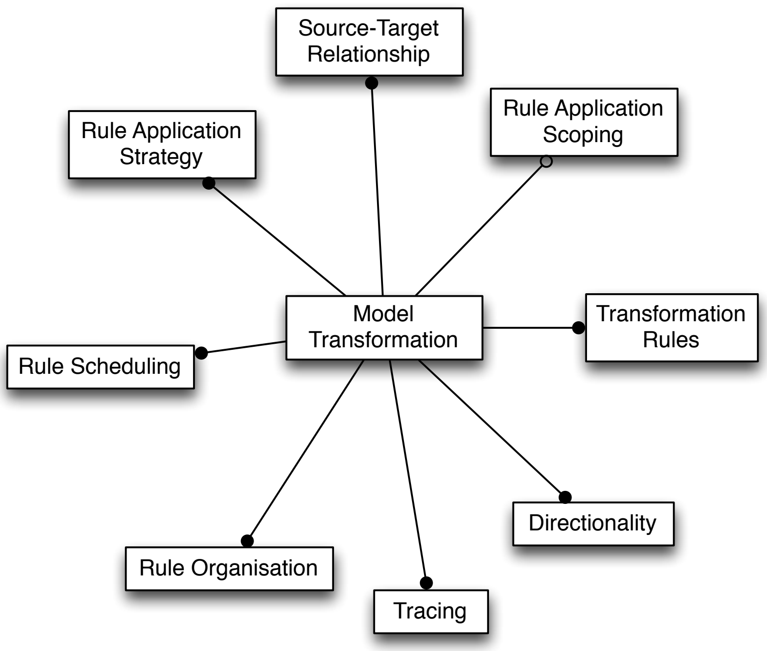

Figure 3.1: Model Transformation Taxonomy

Patterns are in fact model fragments, with zero or more variables. They can take various forms, such as strings, terms or graphs. In *AToM*^{3} and most model-to-model (see **??**) methods, graph-based patterns are used as a visual method to construct transformation rules. Patterns may be represented using the abstract or concrete syntax of the model languages used and this syntax may be either textual or graphical in nature or some combination thereof.

Logic

Logic can express computations or constraints on model elements used in the LHS and RHS of the transformation rule. This logic can be non-executable when it is used to specify relationships between models, but it can also be executable when it is used to retrieve elements from the source model or actually generate model elements when the transformation rule itself is executed.

Transformation rules may syntactically separate the LHS and RHS, but this is not necessarily the case, *e.g.,* when the transformation is executed in-place. Furthermore, transformation rules may be parameterised to alter or fine-tune the rules at the time of their execution.

Figure **??** shows a very simple transformation rule. In this case, both LHS and RHS are explicitly separated and contain models in the Statecharts formalism. The models seen in the LHS and RHS are patterns in the terminology used above, represented in graph form. This very simple rule would connect two unconnected states in a Statecharts model using a transition which fires on the event A.

3.2.2 Rule Application Scoping

Rule application scoping allow us to specify which parts of the model will participate in the transformation. The application scope can be a whole model, but also some subset of the whole model upon which the transformation will be performed.

In graph-based transformation rules, application scoping can take the form of a Negative Application Condition (further referred to as NAC). To continue with the example used in the previous section, figure **??** shows the transition rule from figure **??** augmented with a NAC. The NAC in this example specifies that the transformation can not be executed when the two states occurring in the LHS are contained in a compound state.

3.2.3 Source-Target Relationship

The source-target relationship in a model transformation can take several forms. In the simplest case, the target is a new, previously non-existing model. It's also possible that the target is a pre-existing model and in this case we can differentiate between two scenarios: Firstly, the target might be the source model when the transformation is executed in-place. Secondly, the target might be a different, separate model, in which case the transformation serves to update this pre-existing model.

3.2.4 Rule Application Strategy

When transformation rules are applied to their source models, there may be more than one location in the model where the rule is applicable. In such cases, we need a rule application strategy to specify the where and/or in what order the rule should be applied on the source model.

A rule application strategy can be deterministic, non-deterministic and interactive. Deterministic strategies might follow some well-defined traversal strategy over the source model to iteratively apply the rule to all applicable locations. Non-deterministic approaches on the other hand might randomly select one or more location(s) to apply the rule. Finally, the interactive strategy allows the user to explicitly select where to apply transformation rules, based on manual inspection of the source model.

3.2.5 Rule Scheduling

When a set of transformation rules is available, we need a mechanism to determine the order in which these rules are selected for execution. Rule scheduling mechanisms are responsible for determining the order in which individual rules may be applied. [**?**] specifies four areas in which such scheduling algorithms may differ:

Form

Rule scheduling can be made implicit or explicit to the user. In the implicit case, the user has no direct control over the scheduling algorithm in use. The only way to influence implicit scheduling algorithms is by designing the transformation rules such that a strict order is imposed. On the other hand, explicit scheduling offers the user constructs to control the scheduling of rule execution.

Rule selection

The selection of rules to execute can be performed by some explicit condition or a non-deterministic (random) choice. In both cases, priorities can be associated with rules to enact conflict resolution and/or provide a simple mechanism for ordering rule execution.

Rule iteration

The way in which rule scheduling algorithms may iterate over the transformation rules may differ. Current approaches use either recursion, looping or fixpoint iteration. The latter approach consists of executing rules until no more changes to the target model are detected.

Phasing

Some scheduling algorithms allow the user to organise the transformation process into several phases. Each phase is then associated with a set of transformation rules that may be executed during that phase. For example, a user might c reate some transformation rules for a pre-processing phase of the transformation and other rules to perform the actually desired transformations.

3.2.6 Rule Organisation

Model transformation tools may offer the user the ability to organise the developed transformation rules in some useful way. Just as software systems are organised into logical packages, it is useful to organise transformation rules in a similar way. Mechanisms may be provided to allow a user to package rules into modules. This organisational system may follow the conventions placed forth by the source or target formalisms, but it can also be a completely independent, arbitrary system.

3.2.7 Tracing

Transformation rules can specify links between the source and target models. These links can serve various purposes, the most obvious of which is for identification of model elements between source and target models, such that the transformation is actually applied on the desired model elements from the source model. Figure **??** shows such traceability links in the graph-based transformation system of the *AToM*^{3} tool.

Traceability links may also serve other purposes, they might for instance allow the user to analyse the impact of a transformation on the model at large, be used for debugging purposes, and so forth.

3.2.8 Directionality

Transformation rules can either be unidirectional or bidirectional. Unidirectional transformations are executable in one direction only, source models are transformed into target models. Bidirectional transformations however, can be executed in either direction. The importance of bidirectional transformation lies mostly in synchronisation of models.

3.3 Categories

We have now discussed the various features of model transformation and how these features are the ways in which these currently expressed in model transformation tools. We have cleverly avoided talking about the categories or approaches to transformations that are possible, although we hinted at this in the examples using graph-based transformation rules.

[CH03] defines two major categories for model transformations: *model-to-code* and *model-to-model* transformations. As mentioned in this work, model-to-code approaches are in fact model-to-model approaches where the target meta-model is a programming language and the models are valid programs with respect to that programming language. However, these approaches have some fundamental differences in how the transformation rules may be constructed, so it is useful to look at these approaches as two categories, rather than bunching them together without regard for these differences.

3.3.1 Model-to-Code Transformation

Model-to-code methods, often simply referred to as *code generation*, are used in various scenarios, *e.g.,* to serialise the model to a file on a storage medium, to generate a simulator for the system specified by the model, and so forth. There are two approaches to code generation: visitor-based and template-based. We will use model-to-code transformations to generate the code defined by the Statecharts models created throughout the thesis.

Visitor-Based Approach

The simplest method for model-to-code generation is the visitor mechanism. Here, the internal representation of the source model is traversed and code is written to a text stream, according to the transformation rule defined for the various model elements. This approach is quite similar to the design of compilers in traditional programming languages.

Template-Based Approach

Template-based approaches usually consist of a document in the target language, which is interspersed with metacode referencing the source model in order to generate the required target code. This approach will use executable logic in the LHS of the transformation rules to access the source model and string patterns in the RHS to select code to execute in that location and to perform model expansion. For a more in depth look at template-based code generation methods, refer to [CC01].

Templates greatly resemble the target code to be generated, since the templates themselves are actually 'incomplete' versions of the required target code (it still contains metacode that must be substituted during rule execution). On the other hand, the visitor-based approach is much more declarative and shows little resemblance to the target code. This is not necessarily a positive or negative aspect of either approach, but could be a desired property in some circumstances.

3.3.2 Model-to-Model Transformation

Model-to-model transformation methods concern techniques used to translate a source model to a target model. These models may or may not belong to the same meta-model. There are various model-to-model transformation methods, we will discuss some of the most prominent approaches here, such as: direct-manipulation, relational, graph-transformation, structure-driven and hybrid approaches.

Direct-Manipulation Approach

Direct-manipulation approaches provide the user with some API to manipulate the internal representation of models. The API usually takes the form of an object-oriented framework, which may or may not provide the user with some mechanism to structure transformation rules into logical sets.

Relational Approach

Relational approaches to model-to-model transformation can take various forms, but they are always declarative, with the main idea being that mathematical relations are used to specify the transformation from source to target model.

Graph-Transformation Approach

Graph-transformation approaches operate on typed, attributed and labeled graphs. This is also the method with which model transformations are used in the *AToM*^{3} tool, where the collection of transformation rules are referred to as *graph grammars*. The general notion is that transformations consist of graph patterns on the LHS and RHS of the transformation rule, which may or may not be rendered in the concrete syntax of the meta-model(s). The LHS pattern is matched in the source model and transformed into or replaced by the RHS pattern in-place. In order to provide more control over the matching algorithm, the LHS can be augmented with NAC(s) as was mentioned in the above section on rule application scoping.

Structure-Driven Approach

These approaches typically work in two phases. First, the overall structure of the target model is generated. The second phase then consists of setting all attributes and other properties of the model elements generated in the first phase.

Hybrid Approach

Finally, model transformation mechanisms exist that do not conform to only one approach category. Such transformations use a combination of the above approaches. Examples of such approaches are the Transformation Rule Language proposed in [Omg08] and the Atlas Transformation Language first introduced in [BDJ$^+$03]. Hypbrid approaches are not unusually the consequence of a framework that supports various approaches, in order to allow the user the freedom to choose the approach most applicable to a specific problem.

3.4 In This Thesis

As was previously mentioned, model-to-code transformation shall be leveraged when we compile Statecharts AI models to executable code for use in our test platform. In that case, we will use the pre-existing *SCC* compiler to perform the actual code generation. Model-to-model transformation shall return in the chapter 6, when we construct transformation rules to alter previously created AI models. We will attempt to alter the AI models in an effort to improve their performance with regards to some heuristic(s) and construct a library of model-to-model transformation rules to that effect. Finally, an automated system to apply these transformations will be created, with the goal of allowing model transformation to occur semi-automatically, based on preferences stipulated by the user.

4
Related Work

The work presented in this thesis is for a large part based on previous research and related work. Exploring and understanding this related work has been a large part of the effort to create this thesis; this chapter provides an overview of the work that has served as the basis for this thesis' efforts. Since presenting all details of the related work would deter from the actual goals of this thesis, references to the mentioned works will be placed throughout this paper to aid the reader.

The related work used in this thesis can be divided into two research areas: research towards methods for modelling AI and towards methods for transforming models, more specifically AI models. As such, this chapter is divided into two sections corresponding to these related but different areas of research.

4.1 Modelling Artificial Intelligence

Artificial Intelligence for non-player characters is traditionally developed using a textual programming (or scripting) language. Historically, creating AI for games was rather simple. Sets of rules would be formulated which dictated the actions NPCs were to take if some conditions were met. As the virtual world wherein these NPCs lived became much more complex and dynamic, AI programming needed to evolve. As an illustration of this problem we can take a look at the subproblem of writing pathfinding AI. Writing AI for pathfinding in a static 2D grid-based world is relatively straightforward. Writing the same pathfinding AI for a 3D polygon-based world in which obstacles move about constantly is a much bigger challenge. It is becoming much more difficult to create believable AI than before, because the platforms for which this AI must be written are drastically increasing in complexity.

In large part due to the increasing complexity of virtual worlds, there has been a surge of interest towards modelling AI[1]. Non-player characters are prime examples of reactive systems: They exist solely to react and interact with a player. Current methods for programming NPC AI contain notions of *state* to represent the current condition of characters, *e.g.,* *Idle*, *Fleeing*, *Busy*, ... It seems then that the use of Statecharts to describe the character's behaviour is an interesting area of research. It should come as no surprise that research into this has already been performed. There are alternatives to be

[1]The reasons for this increasing interest is more complex however, as code readability, maintenance, reuse and efficiency all play a big role in this evolution towards modelling rather than programming.

considered however, we will discuss these in section **??**. Let's first take a look at the efforts made towards leveraging Statecharts as a formalism for modelling NPC AI.

4.1.1 Statecharts

Using Statecharts as a formalism to model NPC AI is a relatively new evolution in the game AI world. As such, this practice has so far been largely confined to the world of academic research and no commercial products currently exist that use Statecharts to describe game character behaviour. Most research in this area has been performed at the McGill University in Montreal, Canada under the supervision of this thesis' promotor, Professor Hans Vangheluwe. In the following, we will take a look at the current state of this research.

A first foray in leveraging the Statecharts formalism to describe NPC behaviour was done in 2006 by [DKV06]. In this paper, a method is proposed to allow Statecharts modelling of game character behaviour. The paper served as a precursor to much more detailed [KDV07], in which a much more thorough account is given of the same approach. In these works, the example problem of describing the behaviour of a tank is introduced, which would go on to be the default problem to use as a demonstration platform in further research, including this thesis.

One year after the publication of the above works, in [Bla08], the previously proposed methods are expanded upon even further. This master's thesis was written with the intent to define a way to implement the behaviour of an actual robot using a modelling formalism. It quickly concludes that statecharts is a perfect fit, since many of the formalism's constructs lend itself well to implementing more complex behaviours. It stipulates a couple of requirements for a formalism that is to be used in the context of describing the behaviour of a character in a virtual world.

- *State/Event based:* Our formalism must have a notion of the state of a component and allow for events to be received and broadcast.
- *Autonomous/reactive behaviour:* Components must react to both internal and external events. Events trigger transitions in state, and in turn transitions might broadcast new events to other components.
- *Modularity:* Ease of assembly and reusability of components is highly desireable. The chosen formalism must allow us to reuse components in a different context and provide mechanisms to model our components in a modular fashion.
- *Notion of time:* The current time must retain its meaning in the modelling formalism. The time might be used in the decision to transition from one state of our model to another.
- *Well known:* It is highly desirable for the modelling formalism to have a wide availability of educational material and supporting tools.

So far, both Petri Nets and Statecharts fulfill these requirements and are appropriate formalisms to consider when modelling a reactive system. However, the Statecharts formalism provides us with some extra functionality that aids in modelling behaviour specifically.

- *Composition:* XOR (exclusive-or) decomposition of states, which captures the property that, being in a state, the system must be in only one of it's composite components.
- *Orthogonality:* AND decomposition of states, which captures the property that, being in a state, the system must be in all of its orthogonal components.
- *History:* In the Statecharts formalism syntax, entering a history state is tantamount to entering the most recently visited state.

These constructs greatly aid us in developing behaviour in a modular fashion, enabling component reuse without the need to redesign our implementation. Composition allows us to easily define different behaviour for different conditions, while orthogonality gives us easy access to concurrency, as it might be used in *e.g.,* tracking the state of a lot of different sensors.

While [Bla08] admits there was no real option to choose a modelling formalism (due to the author's recent work in a course at McGill University), we posit that there was no better option than Statecharts. NPCs in games are reactive systems: they respond to the player, other NPCs and their environment, which makes Statecharts – as a formalism designed to be used to describe reactive systems – a natural fit. Furthermore AI for non-player characters is currently often implemented as having a certain state and events are almost always used as the main source of information input. Since these are both syntactic constructs in the Statecharts formalism, we can make use of research already performed on programming AI for non-player characters, without translating the findings to the syntactic constructs of another formalism. While the use of states and events in a textual programming language might not completely overlap with its implementation in Statecharts, the comparison can be made and the use of Statecharts can be seen as an evolution of what is currently accepted as best practice in the field of game AI.

4.1.1.1 Model Architecture

In [KDV07], an architecture for modelling reactive behaviour is first proposed, this architecture is used and somewhat expanded upon in [Bla08]. The architecture is formulated in terms of a *Tank Wars* style game, much like the game used as a test platform in this thesis. Later we will expand upon this work and introduce model transformation as a way to achieve different behaviour from the same source model(s). The proposed architecture is a layered model, where input arrives at one end of the layered stack and output is generated at the other end. The complete architecture is described in Figure 4.1. Note that components towards the ends of the layered architecture – *sensors* and *actuators* have the lowest level of abstraction, they correspond closely to the actual components of the character for which the AI is written. Let's take a look at the various layers and their responsibilities.

Sensors

Sensors are components of the character that deal with keeping track of the current state of the character itself and the character's world. This means we can distinguish between two types of sensors: internal and external sensors. Internal sensors keep the internal state of the character up to date, they concern themselves with *e.g.,* fuel levels, health, energy, ... External sensors on the other hand sam-

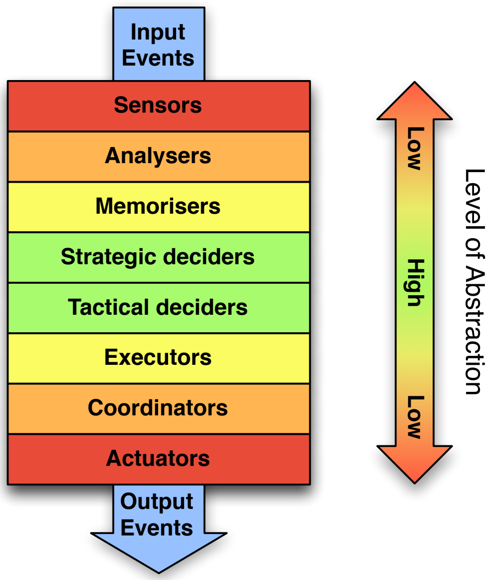

Figure 4.1: Layered AI architecture

ple the world to try and keep track of what is happening around the character. Examples of external sensors would be: eyes (or any other system enabling vision such as radar or cameras), thermometers, altimeters, ...

Each sensor component gets information through (internal or external) events and polling of the game world. This allows sensors to provide an abstract view of the actual state of the character and/or world through the generation of events. In our running tank wars example, the tank behaviour should probably not be influenced by the exact fuel level, but we would like to trigger some kind of behaviour when the fuel level is getting low and/or critical.

Analysers

Not all information needed to describe a character's behaviour can be derived from individual sensors. Sometimes a correlation between various inputs is needed to comprehend the current situation. For example, to determine if some enemy is in range of a character, we need to correlate the information from at least two different inputs. First, we need to find the range of the enemy and secondly we need to correlate this with the range of our character (this usually refers to the range of a weapon carried by the character). *Analysers* will correlate these inputs and generate appropriate events for use in the subsequent layers.

Memorisers

Natural behaviour is not only based on the current state of a system, but is also for a large part based on previous events or states. This implies a need to model the memory of a character. *Memorisers* fulfil this role, allowing us to store important information to reuse at a later time. This information can be stored programmatically for easy access in conditions, but the memory can also be explicitly modelled using Statecharts. In the tank wars example, we could model a system that remembers the last known position of an enemy when this enemy is no longer visible. This information could then be used to navigate towards this location in an effort to find the lost enemy.

Strategic Deciders

Sensors, analysers and memorisers take care of event generation based on the environment, character state and memory. Using this information, the character will plan to achieve certain goals. This is done by the *strategic deciders*. The goals decided upon by strategic deciders should be high level goals. They will therefore correspond to the ’state’ values we often see in current commercial games, *e.g.,* Idle, Attacking, Fleeing, ...

Tactical Deciders

High level goals decided upon by strategic deciders must be translated into lower level commands to the various actuators. This translation is not trivial however, since it might require some complex tactical plans to be made. To facilitate this, the *tactical deciders* layer was introduced. A good example of a tactical decider component is pathfinding. In a certain situation the strategic deciders might decide that fleeing the scene is the best course of action. Without any planning step dealing with how to achieve the ’fleeing’ goal, a character might be able to move away from danger, but this move might bring the character face-first into a wall or into the arms of another enemy. It is clear that more planning is needed. In this case, a tactical decider ’pathfinding’ might exist that can decide waypoints to follow when fleeing, avoiding dangers and obstacles.

Executors

Executors take on the role of mapping the decisions made by tactical deciders to events that can be processed by the actuators. The actual mapping will depend on the facilities offered by the game or simulation platform in question, but typically one executor is used for every available actuator. Returning to the tank wars example, when a tank is fleeing and the tactical deciders have decided upon a route to follow, executors will translate the route to events directed at the engine and steering actuators in order to move towards the current waypoint.

Coordinators

In some cases, decisions made in previous layers will result in incorrect or opposed instructions sent to the actuators. For example, assume a tank is moving forwards with its turret pointed forward as well. If the tank now spots an enemy approaching from its right hand side, it is clear that in order to attack this enemy, the turret must be turned to the right. However, if the tank reaches a waypoint at the same time as the other tank is spotted, and the next waypoint is towards the south-east relative to the tank, the optimal turn for the turret to take in order to take aim at the enemy is no longer a turn to the right, but rather a turn to the left. These cases should be detected and solved by *coordinators*.

// TODO clarify this using an illustration.

Actuators

Actuators perform the actual commands that are the result of decision made by previous layers. Events taken as input by actuators should be relatively simple commands, such as *speedup*, *turnleft*, *fireturret*, etc.

4.1.1.2 Mapping To A Platform

The model architecture explained above results in a complete description of the behaviour of a non-player character. Since we are using the Statecharts platform, the architecture is completely event-based, meaning that time is regarded as being continuous and events can occur at any time. Modelling the behaviour in continuous time has several advantages according to [KDV07]:

Model freedom
The modeller is not constrained by implementation issues and can focus on the required behaviour logic.

Symbolic analysis
The model can be analysed using timed logic to prove properties.

Simulation
Simulation can occur with infinite accuracy in simulation environments.

Reuse
Continuous time is the most general formalism possible and can thus be used in any simulation environment.

When the Statecharts AI models we create are used in some environment, actual code must be generated for execution in that specific environment. When using AI models on event-based platforms, there is no issue. We can use the models as they are, feeding input events to the sensor layer and processing events generated by the actuator layer. The situation becomes less obvious when the destination platform is time-sliced, as is the case in most game engines. When the platform is 'time-sliced', we mean that time does not advance continuously but in small, discrete steps of *e.g.,* 50 ms. For such platforms, we will need to bridge the event-based – time-sliced gap.

To enable event-based reasoning in a time-sliced environment, we will need to perform some logic to bridge these two paradigms. [KDV07] proposes the use of a function *void AI(tankAIInput, tankAIOutput)*, in which the variable *tankAIInput* is some data structure containing the state of the tank at time of the function call and *tankAIOutput* contains the state of the tank after the model logic has been executed. This function then proceeds by copying the appropriate values to the tank sensors and subsequently evaluating the guards of transitions, firing their corresponding events if applicable. Both [KDV07] and [Bla08] propose mapping to events at the sensors layer. From the sensor layer onwards, all events are propagated and actions are triggered within the Statecharts formalism. The last step is then copying the appropriate values from the actuator layer back into the tankAIOutput variable.

It is important to note that both [KDV07] and [Bla08] approach this mapping from the viewpoint of the EA Tank Wars [Art05] platform. This platform was used in the context of a competition where students would create AI for tanks and let their creations compete against each other. This implies that the platform was heavily constrained, in the sense that the AI had no direct control over the various components of the tank, but the state had to be communicated through a software layer that checked against cheating in some way. In this thesis we explore how we can use Statecharts to model AI in any game platform, which implies the AI has the ability to directly control various components of the characters. We will explore how to exercise this control in chapter 5.

4.1.2 Decision Trees

Decision trees have been used for creating AI in games for a while and the approach is still gaining in popularity. They provide a visual system to specify how decision should be taken by non-player characters. An simple example of such a decision tree can be seen in figure **??**.

While decision trees do indeed provide a visual formalism to model some key aspects of non-player character behaviour, the approach is effectively no different than a nested if-construct in a textual programming language. It offers no real benefits other than the ability to visually edit behaviour and as a consequence, the approach is equivalent to using a textual programming language to specify which behaviour should occur under which circumstances. Our approach on the other hand does not concern itself with which behaviour needs to be observed, but rather with the consequences of external and internal stimuli on the internal state of the character, which in turn results in some actions being taken. Our approach thus lends itself well to emerging behaviour rather than scripted behaviour as is the case with decision trees.

4.1.3 Other notable projects

There are other approaches to modelling behaviour of non-player characters, but nearly all of these concern themselves with scripting the behaviour to achieve a cinematic experience, as opposed to modelling the actual character such that emergent behaviour can be observed.

The *Unreal 3 Engine* [Inc12c] for example offers users the Unreal Kismet visual scripting formalism. Kismet can be used by 'artists and level designers' to control 'how a level will play without touching a single line of code', according to the official website. It allows developers to formalise how certain scenes will play out, essentially providing a visual formalism for scripted actions. While this allows non-programmers to create scenes, it does not in fact concern itself with modelling a reactive system.

Another example is *Simbionic* [FHJ03], which is a set of tools allowing developers to describe the behaviour of intelligent agents using finite state machines. This premise shows remarkable parallels with our approach, but some important differences must be noted. For one, the state of agents is represented as actions. Transitions from one action to another is done using conditions and guards. Furthermore, the framework operates exclusively in a time-sliced fashion, not allowing for the abstraction of time to continuous time when modelling an agent's behaviour. Lastly *Stottler Henke Associates, Inc*, the company behind the Simbionic project, has not released any new research since 2003, indicating that the drawbacks of the Simbionic framework are not expected to be addressed anytime soon.

4.2 AI Model Transformation

Despite the fact that model transformation is a very broad area of research, there has not been a great deal of research invested in transforming AI models. This should not come as a surprise to the astute reader, since the practice of visually modelling AI itself has not gained widespread usage at this time.

Transformation of AI models is most useful for the generation of variants from a certain starting model. The most advanced research in this line of work comes from the *Modelling, Simulation and Design Lab* at the McGill University. The last few years, work has been performed in the area of procedural generation of Statechart AI. In *Generating Extras: Procedural AI with Statecharts* [DKV11], the authors lay the groundwork for procedural AI generation by transforming AI models made using the Statecharts formalism.

Generating variants is not the only use for model transformations in the context of game AI, however. Altering AI models such that some sort of desired behaviour results from a (sequence) of transformation is another useful application. The desired behaviour can take many forms. Model transformation can be employed to generate enemy characters who are 'just right' for some player, posing the player with a challenge which is at the same time difficult, but easy enough to still be entertaining. It can also be used to find the 'best' AI for a specific task. As a matter of fact, creating a 'pleasing' game experience can be viewed as a task in this context, and finding the 'best' AI for a task can be determined by some heuristic that evaluates the AI's performance. Unfortunately, the research that has been performed in this area is exceptionally limited, a short overview of this work will be given in section 4.2.2.

4.2.1 Procedural AI Generation

The authors of [DKV11] limit themselves to the introduction of variation and personality to pre-existing AI. This pre-existing AI is modelled using the layered Statecharts architecture outlined in the previous section. Three methods for introducing this variety are proposed: varying parameter values, component configurations and component models.

4.2.1.1 Varying Parameter Values

Non-player characters have many properties defined for each instance of the NPC. Examples of NPC properties are *health*, *fuel level*, etc. Varying these properties for each new instance is an obvious way to introduce variety in a group. The proposed technique associates valid *ranges* with each numerical property, allowing an algorithm to choose a value in this range conforming to some specified *probability distribution*. However, since some properties might depend upon others, the need for *dynamic ranges* arises.

Let's take a look at a simple example character to better understand this technique. Consider an enemy soldier NPC with two properties: *weight* and *fitness*. Weight might influence how the physics reacts to the NPC's presence and fitness might be a property used to determine how long the character can sprint or take other actions. It is clear that in the real world, a certain correlation exists between a person's weight and their level of fitness. Assume we associate the range $[60kg, 90kg]$ with the weight property and $[0.5, 1.5]$ with the fitness property (if a value of 1 indicates a 'normal' fitness level). It would then be possible that characters are generated with a high weight and high fitness property, which might not be desirable. It would be better to relate these two properties in some way, such that the fitness range will be higher or lower depending on the character's weight. We could define the fitness range as follows:

$$
\begin{aligned}
weightRange &= [60, 90] \\
fitnessRange &= [0.5 + weightDeviation(), 1.5 + weightDeviation()]
\end{aligned}
$$

where

$$
weightDeviation() = \frac{minWeight - weight + 0.5(maxWeight - minWeight)}{maxWeight - minWeight}
$$

This new, dynamic range for the fitness property ensures that heavy characters get a possible range for the fitness property that is much lower than the possible range for lighter characters. In this simple example, a character with a weight of 60kg would have a possible fitness range of $[1, 2]$ and a weight value of 90kg would allow a fitness range of $[0, 1]$.

The existence of dynamic ranges also implies a need for organising the evaluation of the various properties, all dependencies must be analysed such that no unresolved values for properties occur when evaluating the ranges. Additionally, we'll need to check for circular dependencies and erroneous values. The complete proposed method for varying parameter values is given in listing 4.1.

```
1. For each parameter:
   2. Create a range for each parameter.
   3. Decide on the probability distribution best suited
      for that parameter.
4. Examine parameters for dependencies.  For each
   dependency:
   5. Determine if the dependency is critical.
```

```
6. If critical, assign dynamic ranges to resolve
   problem.
7. If non-critical, resolve if desired, perhaps using
   dynamic ranges.
```

Listing 4.1: Proposed method for varying parameter values

4.2.1.2 Varying Component Configurations

The previous method proposed varying the parameter values with the intent of achieving different behaviours for *individual components*. This section presents a method for changing the *overall behaviours* of an AI by assembling existing components in different configurations.

Removing Components

The simplest method for generating new component configurations is to remove components. Because our model architecture implies a loose coupling between components, this is relatively easy to do. Events are generated without regard for the components that will react to these events and reaction to events is done without regard for the component(s) that might generate them. Theoretically, for an AI consisting of n components, 2^n different component configurations are possible through removing components. However, in practice this is not the case. Often times, there is only one strategic decider component. This layer functions as the connection between the input and output of the AI and severing this connection would lead to an AI that is unable to react to any input. Furthermore, removal of sensor and actuator components is dangerous, because they form the link between the AI and the outside world. This could potentially be leveraged when *handicaps* are desired (no vision, no movement, no hearing, etc.).

Replacing Components

Another method is to replace components with other equivalent components. Components in our model architecture are loosely coupled with the classes for which they define behaviour, allowing us to do one of two things. We could associate an existing class with a new Statecharts model or the opposite, create a new class and associate it to an existing Statecharts model. There is one caveat to this: components might have semantic equivalence, but not necessarily syntactical equivalence. This means that a replacement component might not generate the events that are expected by components in the following layers. So in order to automate generation of new component configurations using replacement of components, we need to create a library of semantically equivalent components and a mapping to specify the correspondence between events.

Adding Components

A final method consists of adding new components to an existing configuration. Adding components to the initial layers – sensors up to strategic deciders – is quite easy. Adding new sensors allows the AI to react to new data about the world and new analysers could improve the AI by detecting new higher level events from a correlation of lower level events. Adding components to the later layers – strategic deciders up to actuators – is somewhat less trivial. These layers react to events generated by earlier layers, so an introduction of a new component at these layers typically means more steps are required to actually utilise this new component, such as event renaming.

[DKV11] further states that the true power of varying component configurations is to be found in the combination of the above methods. Combining event renaming and component addition allows for completely new types of behaviours that work together well with the existing configuration.

4.2.1.3 Varying Component Models

Arbitrary structural modifications of the AI behaviour models is the most drastic approach to altering the described behaviour. [DKV11] suggests modelling such modifications explicitly using transformation rules, as discussed in chapter 3. Specifically, rule-based model transformation using graph transformations is employed. Some typical modifications applied using this method are *resetting components* (enclosing a state in a superstate with a transition to itself on some event or interval) and *introducing orthogonality* (moving a global state into an orthogonal component). These kinds of modifications perform structural changes that are not possible through other methods.

4.2.2 Determining Good Transformations

The introduction of this section hinted at the possibility of finding an AI that is the 'best' at a certain task through transformations. This goes a step further than the previous section about procedural generation, since we now want to generate variants with some sort of desired behaviour in mind. There has not been extensive research in this field, marking this thesis as the first to attempt to automate this process. There has, however, been some research towards automatic customisation based on the actions of a player in the game.

Let's first have a look at how this is currently done in commercial products. Here we can separate games into two large categories: *level-based* and *non-level-based* games.

Level-based

This category contains all games in which the player has a certain level, which usually starts at 1 and may or may not have an upper limit. Arguably the most popular example work in this category is *World of Warcraft* [Inc12b], which brought the concept of character levels to the attention of a broader audience and heralded its introduction in games outside of the RPG[2] genre. The level of a player determines his or her strength, as relating to any part of the game. This often means that as players play the game, their level rises and enemies become increasingly easy to defeat. Some games assign a level to enemy characters to, allowing the enemies to level up together with the player. In effect, this creates a sort of binary playing experience. Either enemies are too hard for the player's level or they are too easy. Only when following a predetermined path through the game – usually corresponding to the main story line – will enemies

[2]Role Playing Game

be perfectly tailored to the player. A lot of effort has gone into this system in the last few years, some games apply levels better than others, but overall this system is recognised as being too superficial to create a realistic notion of difficulty.

Non-level-based

Any game in which the player does not have a level belongs to this category. There are various ways in which games try to offer an interesting difficulty curve to players without introducing levels: learning new abilities, increasing enemy character's damage done, etc. This creates an experience that evolves in blocks, each time something new is introduced to alter the difficulty of the game the player's experience changes. If this is executed with care, it creates very interesting dynamics between the player and the enemies. However, there is no real alteration in behaviour for the enemies. Some enemies are more powerful than others, but on the whole, no significant change in behaviour is introduced. A popular example for this category is *Bioshock* [Sof08], which introduces the player to new abilities, enemies and environments in order to create a difficulty curve that feels naturally tailored to the player, but is actually set in stone.

Both categories of games do not actually look at how the player is playing the game in order to determine how the enemies should act. A skilful player could therefore rush through the first levels, where a less experienced player might have a hard time advancing. We must note that some games do try to solve this problem. A popular example is the *Call of Duty* [Inc12a] series of games. In the last iterations, these games offered a training level in which the player's performance is assessed. Depending on the assessment, one of the available difficulty levels (easy, normal or hard) is suggested. It's obvious that such broad difficulty levels can impossibly tailor the experience to every player.

Automatic Transformation

Recent work has tried to offer ways to tailer the whole experience to the individual player. In [Li08b] and [Li08a], the possibility of automatically customising NPC's based on a player's temperament is discussed. While this thesis does not concern itself with determining a player's status in order to transform NPC's behaviour, this related work nonetheless offers some good insights in how variations can aid in creating an optimal game experience. The author proposed to present a questionnaire to the player when a game is started. The analysis of this questionnaire will then prompt certain variations to arise in the NPC behaviour. This questionnaire can be presented at regular intervals, to allow the experience to keep evolving as the player's temperament might evolve. The actual variations are performed using transformation rules on a Statecharts AI model. While some questions may be posed concerning the intrusiveness of presenting the player with a questionnaire before gameplay can start, this work shows that determining a player's abilities and/or state of mind is a very helpful tool to determine the kind of variations that are desirable. The specific model to use when describing the various aspects of an individual player remains an open question.

5
Artificial Intelligence Modelling Framework

6 Artificial Intelligence Model Transformations

7
Integration into a Commercial Product

8
Conclusion

Bibliography

[Art05] Electronic Arts. Ea tank wars. `http://info.ea.com/company/company_tw.php`, 2005.

[BDJ$^+$03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the atl model transformation language: Transforming xslt into xquery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[Bla08] Silvia Mur Blanch. Statecharts modelling of a robot's behavior. Master's thesis, Escola Tècnica Superior d'Enginyeria Electrònica I Informàtica La Salle, 2008.

[CC01] Craig Craig Cleaveland and J. Craig Cleaveland. *Program Generators with XML and Java with CD-ROM*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[DKV06] Alexandre Denault, Jörg Kienzle, and Hans Vangheluwe. Model-based design of game ai. 2006.

[DKV11] Christopher Dragert, Jörg Kienzle, and Hans Vangheluwe. Generating extras: Procedural ai with statecharts. March 2011.

[FHJ03] Daniel Fu, Ryan Houlette, and Randy Jensen. A visual environment for rapid behavior definition. In *In Proc. Conf. on Behavior Representation in Modeling and Simulation*, 2003.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[Inc12a] Activision Publishing Inc. Call of duty®. `http://www.callofduty.com/`, 2012.

[Inc12b] Blizzard Entertainment Inc. World of warcraft. `http://us.battle.net/wow/`, 2012.

[Inc12c] Epic Games Inc. Game engine technology by unreal. `http://www.unrealengine.com`, 2008-2012.

[KDV07] Jörg Kienzle, Alexandre Denault, and Hans Vangheluwe. Model-based design of computer-controlled game character behavior. In Gregor Engels, Bill Opdyke, Douglas Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 650–665. Springer Berlin / Heidelberg, 2007.

[Li08a] Kyle Li. Project report: Automatic behavior customization of non-player characters using players temperament. 2008.

[Li08b] Kyle Li. Reading report: Automatic customization of non-player characters using players temperament. 2008.

[Omg08] Qvt Omg. Meta object facility (mof) 2 . 0 query / view / transformation specification. *Transformation*, (April):1–230, 2008.

[Sof08] Take-Two Interactive Software. Bioshock. `http://www.2kgames.com/bioshock/`, 2008.