Building Robust Real-Time Game AI
Simplifying & Automating Integral Process Steps in Multi-Platform Design

Gaudl, Swen

Award date:
2016

Awarding institution:
University of Bath

Building Robust Real-Time Game AI: Simplifying & Automating Integral Process Steps in Multi-Platform Design

submitted by

Swen Gaudl

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Computer Sciences

May 2016

Acknowledgements

After nearly four years of work and continuous learning, reading, programming and experimenting and iterating this cycle, the thesis is finally complete. Driven by my desire to understand and advance the development of agents for highly demanding domains such as digital games, I explored many scientific fields. I ventured into robotics, cognitive science and biology to understand modelling approaches for sophisticated agents. Now, this journey comes to an end, and the cumulated knowledge I gained is collected within this thesis.

During those four years, I could concentrate full-time on my research due to the support I received from the University of Bath and the Department of Computer Science. Due to my University scholarship, I could focus and dive deeper into research than I initially had hoped. I enjoyed teaching and the interaction with graduate and undergraduate students. I was able to visit the University of California, Santa Cruz, because of the University travel grant which I am thankful for. I would also like to thank my supervisor Joanna J. Bryson for giving me the opportunity to undertake my studies in the first place and also for initiating my research visit.

I would like to thank Michael Mateas and Noah Wardrip-Fruin for integrating me into their research group for that short period of time and for spending the time to discuss my research. I want to thank April Grow, Joseph C. Osborn for their collaboration and time. Additionally, I want to thank James Ryan and his wife Nina for taking care of me in Santa Cruz; James on top of that for our research discussions and his feedback on my research. I wish I could have stayed longer. They and all other researchers I met at UCSC allowed me to expand my understanding on games and games research a lot.

I would like to thank Leon Watts, a lot, for our continuous conversations on research over coffee and tea and for listening and discussing my research ideas in great detail. I want to thank Rob Wortham and Andreas Theodorou for their feedback and our conversations on action selection and the philosophy of computation and complexity.

A special thanks to Paul Rauwolf and Dominic Mitchell for our long conversations and our discussions on research rigour and how to survive a PhD.

I am also grateful for the tremendous amount of work Eliza Shaw and Michael Wright undertook, helping me to enhance the understanding and readability of my thesis. I do feel that my writing skills got better because of you and your feedback.

I am forever indebted to my partner Denise Lengyel, especially for the last eight months when I spent most of my time writing, eating and brooding over this work ignoring most other things. Without our conversations and your support and patience, this would not have been possible. I would like to thank my family for not giving up on me when I was too busy to reply or answer calls and for their moral support and encouragement.

Lastly, I want to thank the numerous people I did not manage to name and those I met during my PhD at conferences and workshops which gave feedback to my research and thus took part in shaping it.

Summary

Digital games are part of our culture and have gained significant attention over the last decade. The growing capabilities of home computers, gaming consoles and mobile phones allow current games to visualise 3D virtual worlds, photo-realistic characters and the inclusion of complex physical simulations. The growing computational power of those devices enables the usage of complex algorithms while visualising data. Therefore, opportunities arise for developers of interactive products such as digital games which introduce new, challenging and exciting elements to the next generation of highly interactive software systems. Two of those challenges, which current systems do not address adequately, are design support for creating Interactive Virtual Agents (IVAs) and more believable non-player characters for immersive game-play. We start in this thesis by addressing the agent design support first and then extend the research, addressing the second challenge. The main contributions of this thesis are:

- The POSH-SHARP sytem is a framework for the development of game agents. The platform is modular, extendable, offers multi-platform support and advanced software development features such as *behaviour inspection* and *behaviour versioning*. The framework additionally integrates an advanced information exchange mechanism supporting loose behaviour coupling.

- The *Agile behaviour design* methodology integrates agile software development and agent design. To guide users, the approach presents a work-flow for agent design and guiding heuristics for their development.

- The action selection augmentation ERGo introduces a "white-box" solution to altering existing agent frameworks, making their agents less deterministic. It augments selected behaviours with a bio-mimetic memory to track and adjust their activation over time.

With the new approach to agent design, the development of DEEPER AGENT BEHAVIOUR for digital adversaries and advanced tools supporting their design is given. Such mechanisms should enable developers to build robust non-player characters that act more human-like in an efficient and robust manner. Within this thesis, different strategies are identified to support the design of agents in a more robust manner and to guide developers. These discussed mechanisms are then evolved to develop and design IVAs. Because humans are still the best measurement for human-likeness, the evolutionary cycle involves feedback given by human players.

Related Publications

- Grow, A., Gaudl, S. E., Gomes, P. F., Mateas, M., and Wardrip-Fruin, N. (2014). A methodology for requirements analysis of ai architecture authoring tools. In *Foundations of Digital Games 2014*. Society for the Advancement of the Science of Digital Games

- Gaudl, S. E., Davies, S., and Bryson, J. J. (2013). Behaviour oriented design for real-time-strategy games – an approach on iterative development for starcraft ai. In *Proceedings of the Foundations of Digital Games*, pages 198–205. Society for the Advancement of Science of Digital Games

- Gaudl, S. E. and Bryson, J. J. (2014). Extended ramp goal module: Low-cost behaviour arbitration for real-time controllers based on biological models of dopamine cells. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE

- Gaudl, S. E., Osborn, J. C., and Bryson, J. J. (2015). Learning from play: Facilitating character design through genetic programming and human mimicry. In *Progress in Artificial Intelligence*, pages 292–297. Springer International Publishing

Contents

Chapter 1

Introduction

In this thesis, we seek to understand and advance the current state of the art in artificial intelligence (AI) design and development approaches for controlling embodied agents. To achieve this goal, a comprehensive literature survey was conducted which is backed up by expert interviews. Based on the findings, a new architecture and methodology for agent design are proposed—Agile Behaviour Design, discussed in Chapter 5. To support the design methodology, a new agent framework is proposed to allow for experimentation and testing of the new methodology. This new agent framework is posh-sharp which integrates novel mechanisms for inspecting and versioning of behaviours and supported by a new arbitration mechanism—ERGo, discussed in Chapter 6—which responds significantly better in noisy environments than traditional approaches.

1.1 Motivation

To pursue the goal of a more robust and supportive agent design environment, while not arriving at a too abstract answer to render it unusable, a special focus is put on behaviour-based AI (BBAI) [Maes, 1993; Brooks, 1986] for development and design of character artificial intelligence. As embodied agents interact in an environment, digital games are chosen as an interesting and challenging spatial environment for the character to interact in.

Existing character design tools, methodologies and architectures were analysed to understand their design process and differences in approaching agent design. Based on this analysis, the most fitting methodology was selected and enhancements were proposed to it. Additionally, an entirely new combination of tools is presented which results in a usable toolbox for authors of digital games in combination with the given

methodology. The importance of "academic" game design tools is often viewed with suspicion from the industry. This is not only due to the disparity in expectations on stability and robustness but also due to the fast turnover in the industry when it comes to new tools and techniques.

However, there are a variety of reasonable arguments for pursuing this work. The first being that the architecture this work is built upon, the Parallel-Rooted Ordered Slip-Stack Hierarchical (POSH) planner [Bryson and Stein, 2001], is largely similar to the dominant approach BehaviorTree (Bt) introduced by Isla [2005] to the games industry. Creating a comparative approach makes the knowledge transfer from one approach to the other easier. The second is that the focus on the agent design process and tools for developing agents is relatively new to the domain of game AI, which itself is a relatively young research field in academia as Laird and van Lent [2000] show. A third important point this work tries to address is the focus on both sides of the research, the technical architecture and environment as well as the academic research supporting its merit.

The decision to focus on behaviour-based AI was deliberately made due to the interesting property of BBAI, namely the disparity between the understanding and modelling of living entities, such as animals, and the modelling of intelligent artefacts, such as robots or embodied characters can be studied and observed explicitly. The purpose of modelling living entities is to stay as close as possible to important characteristics of the original to create a meaningful representation of their features and abilities. However, when creating intelligent artefacts no such limit exists. To go even further, being bound by implying artificial limits from living entities might even restrict the resulting creation in an unnecessary or non-beneficial way.

Using digital games as an environment is advantageous when trying to advance AI design and IVA as games not only provide an abstraction of physical environments but are also accessible to a wide audience, thus, making testing and evaluation easier. Looking closer at one aspect of games, namely the abstraction from a real physical environment, allows us to remove the problems physical sensors and agents bring with them and focus on a higher level of development. Using a robot as our target host would require an additional focus on lower levels of the robotic system to tackle inaccuracy in sensors or malfunctioning sub-components which generate a high amount of noise on the lowest layers.

First and foremost, the knowledge gained from creating a toolbox for developing character design tools for games is not limited to the development of agents for said environment. This knowledge can be further abstracted, thus, most of the tools can

also be used for other environments where artificial agents[1] are employed. Applicable domains for those new tools range from consumer and care robotics such as PARO [Broekens et al., 2009] to assistant systems.

The next section discusses the current game development process focusing on the specific parts which are crucial for agent development. This leads to identified critical points in of the development process in Chapter 1.3.

1.2 How to Build Games and Agents Therein?

To understand and explore the domain of this work—character AI for digital games—let us start by understanding the broader domain of digital games first.

1.2.1 Game Development

As an initial definition, digital games are highly interactive software systems. The general assumption on digital games development is that the development process is similar to that of other software systems, e.g. we treat their development in similar ways to the development of control software for an elevator. However, most successful game developers [Bethke, 2003; Keith, 2010] employ approaches mixing traditional software development with approaches from film or TV production based on the creative aspect of the resulting product. A closer examination of the game development literature in industry reveals that there is a widespread assortment of approaches due to the interdisciplinary nature of the parties involved in the development and production of a game. There are sparse academic contributions to this domain due to the perceived accessibility and financial risk of involving research in large scale games projects. There are, however, two major streams which examine game development:

- From an artistic or creative perspective [O'Donnell, 2012; Zackariasson, 2013]: This results in processes similar to film and movie productions.
- From a technical perspective: The resulting product *is* a software product, thus, the project is treated as one, to manage the completion of the development [Chandler, 2009; Bethke, 2003].

Additionally, there exists a broad range of literature such as the Game Development Principles by Thorn [2013] which does not reflect on the different aspects of a game or advances made in software development. Rather, it is creating an inaccurate and

[1] A definition of what an agent is follows in the next chapter.

sometimes misleading impression of the development process. An example is the presentation of a proprietary project management process when guiding the reader rather than presenting or comparing it to the industry standards.

The first group highlights that game development is not software development but a more creative process [O'Donnell, 2012]. O'Donnell states that game developers need to put more focus on the creative side to not be bound by too restricted conventional software development:

> "Although video games are software, they are more than software, and too often they and their producers remain lumped into the same category as software developers. Assumptions are made about what compromises a game and its production process, which continues to hold the video game industry and the art of game production back from its full potential. [...] Perhaps most importantly, the gross mislabeling of video games as software and game development as software development significantly distorts the creative labour ..." [O'Donnell, 2012, p.30]

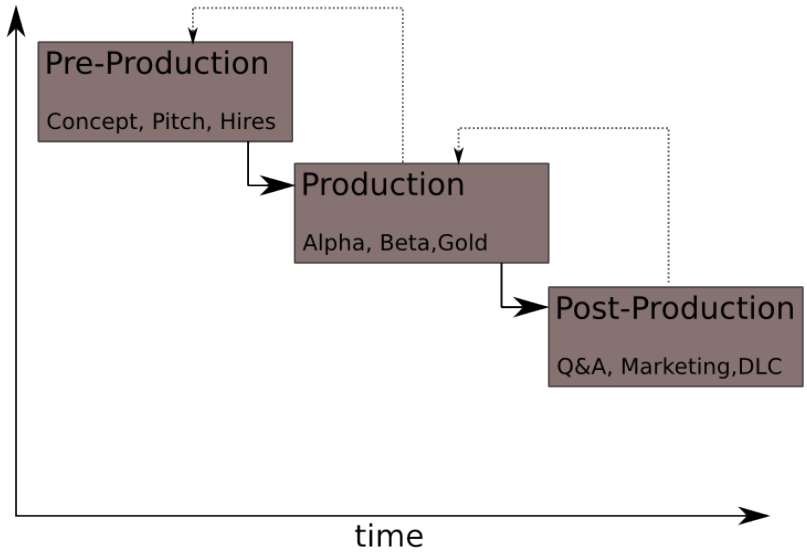


Figure 1-1: An Iterative Waterfall process model used in the games industry. The model contains the three phases: Pre-production, Production and Post-production. During each phase there is a way to step back to the previous phase to correct mistakes or include new requirements.

Based on the evolution of the games market and its origins rooted in software development, all game development approaches have three essential phases in common:

- **Pre-Production:** This phase is sometimes split into two parts, concept phase and pre-production. The split is based on the fact that after the initial idea brainstorming and game concept, the game design will be proposed or pitched to a publisher as a new potential project for funding which can result in two outcomes. If the publisher is interested and supports the proposal, the project receives the "green light" for funding. This statement of support puts the producer in a position to continue and officially work on the project. The team is now officially built, and game designers extend the concept into a game design. If no "green light" is given, either the project is scrapped, or the concept and proposal will be re-worked for another publisher.

- **Production:** During this phase the game design is still tweaked and is used in turn to develop the game. The game development is split into at least three phases:

  - Alpha—during this phase a first version of the game is implemented.
  - Beta—during this phase the game is changed based on the test results of the Alpha and features are stabilised.
  - Gold—the game is completed and thoroughly tested and prepared for distribution.

- **Post-Production:** Testing and Q&A is still carried out during this phase to fix late issues. The game box and packaging, or the online store page are made ready for distribution; in parallel, the marketing for the product is done. The development work is shifted towards Downloadable Content (dlc), if applicable.

This description is based on the classical model of the games industry for large and expensive projects. Thus, the staged process does not directly reflect on how crowdfunding projects or projects by independent developers (indie developers) are structured [Martin and Deuze, 2009]. For some of the latter, the publisher is replaced with a crowdfunding service such as kickstarter[2] or indiegogo[3], or the project is self-funded; thus, the publisher is replaced or completely removed at the pre-production and production phase.

[2]Kickstarter is an American-based community funding portal which is actively supporting the development of projects including games. A project which aims for funding needs to set a funding goal. Only if the funding goal is reached, the project will receive support from the community. The portal is accessible at:`https://www.kickstarter.com/`

[3]Indiegogo is similar to Kickstarter, but additional includes pure fund-raising and does not require a funding goal to be reached to allow the project to continue.The portal is accessible at:`https://www.indiegogo.com`

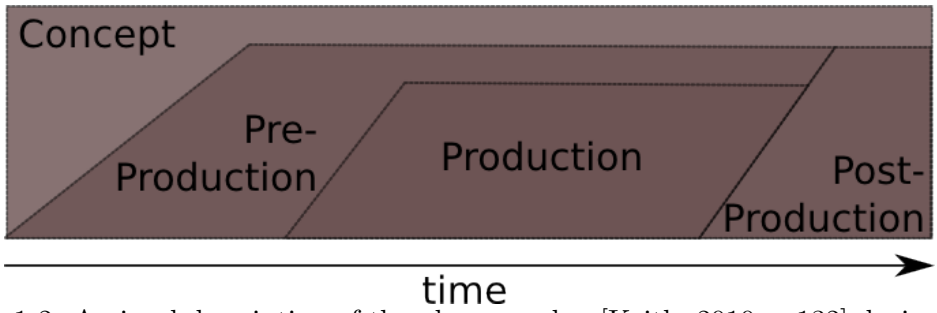


Figure 1-2: A visual description of the phase overlap [Keith, 2010, p.132] during game development. Concept and Pre-production in this illustration are separate phases to illustrate how long adjustments are still made to the design.

Most approaches used in the games industry [Rabin, 2010b; Thorn, 2013; Rubin, 2012; Harbour, 2004] separate the phases as mentioned above and put them in a fixed sequential order which creates a waterfall-like process model, see Figure 1-1. Based on extensive experience Keith [2010] promotes a form of software development which requires a slightly adjusted development process as illustrated in Figure 1-2. Since the introduction of more agile software development models such as Scrum [Rubin, 2012; Keith, 2010] in the early 2000's, developers are in a position to be able to adjust the development and the design process more flexible, in contrast to the conventional waterfall model. This allows developers to include new features and requirements dependent on the overall progress of the development, without breaking the process model.

A comparison of the arguments made by O'Donnell [2012] and Rubin [2012] reveals that one issue, namely "Game development is not just developing software but also creating an entertainment medium", is not properly addressed. This leads to the question that asks, if it is not more beneficial to investigate alternative development approaches instead of the standard software development approach. However, O'Donnell does not provide an actual development model. In contrast to this, Keith includes the communication, design, and testing based on Scrum sprints and milestones into an agile iterative process.

Let us examine the three phases of game development and the overlap which Keith [2010] states. During Pre-production, a concept is created for the game including some of the essential elements which should make the future game unique. Typically, only a prototype is used to illustrate some the most important ideas. A game design document is created which will be used during the whole development to track the progress and divergence from the original idea. The development of the actual product commonly starts during the production phase. During this phase, the game design

gets translated into software. Adjustments to the document are made as well, based on tests and evaluations of parts of the game undertaken by the game designers. This phase, as already discussed, is also split into at least three sub-phases. These phases track the development milestones of the game. After each of the three phases, the game is typically play-tested to gather feedback on playability, bugs and enjoyment of the product. This testing process is quite expensive as it requires a large amount of time, data analysis and recruitment of testers from the targeted audience. Testing also bears the risk of leaking early information or prototypes of the game, problems which can result in undesired early or negative attention to the game. After the Gold release, the game will be made ready for distribution. As the game is in its final production state, no changes are made to it anymore, however, testing and Q&A are still conducted. All new and upcoming changes are now pushed into a first patch which will be released either on day one of the release or at a later point. For larger projects the work also starts to shift towards additional content for the game or towards a follow-up project.

Earlier the comparison of games as traditional software systems was made, suggesting to treat games in a similar way to control software for an elevator. In most of the aspects of development such as when it comes to writing the software and managing the overall milestones of the project, this comparison holds. However, due to the amount of creative input and the inclusion of diverse and qualitative user feedback based on implicit criteria such as fun or engagement, games require a different form of attention or process.

1.2.2 Developing Game Agents

The actual game AI development starts in the production phase when a team agrees on the used technology, such as a game engine or external tools and modules. In some cases, parts of the AI system could already be developed at the end of pre-production when an early prototype is built to demonstrate how the game should work. Due to the conceptual design, game characters can already be part of the game design document if the designer is focusing on story and non-player characters (NPCs) as well as player interaction in their design. Thus, the character AI system either needs to be considered at that point or the AI system needs to be modular and flexible enough to deal with all, or most, potential requirements. For some projects technology is developed from scratch, similar to large movie productions.

Harbour provides five points when developing AI for games [Harbour, 2004, p.581]:

1. It is enough to make AI which looks intelligent. *[Facade or Impression suffices]*
2. Design the AI by putting yourself into its position. *[Ego-centric Design]*

3. Apply the simplest feasible technique. *[Simplicity]*

4. Pre-design the AI and its behaviour. *[Modelling behaviour]*

5. Use random behaviour as a fallback if no other approach works. *[Feasibility]*

Taking Harbour's expert knowledge as a basis, the best potential approach is using a system the design and programming team are familiar with or which has a shallow learning curve, is extensible and designer-friendly. This is supported by Rabin [2010b], he discusses in his Chapter 5.3 what game AI should encompass. He also extends the rules made by Harbour stating that game AI should be as intelligently looking as possible but should contain an exploitable logical flaw to challenge the player. Rabin also stresses the requirement on CPU time game AI is supposed to use—less than 20% of the time per frame. Another important point is that the AI system should never be able to affect negatively the development time of the game. Thus, it must be a robust system which can scale up to the requirements for complex agents.

Due to the different skill sets required for making games the person implementing the game AI is rarely the person designing the game characters or the narrative of the game. In larger teams specific game or level designers[4] design and develop characters and how the player interacts with the game. These designs are normally not done directly in the program code but with the use of other tools, such as storyboards, use cases, textual descriptions. They add to parts of the game design document and are used as references for implementation. In some companies, the game character design is further supported by more visual tools which are either developed in-house or licensed for development. In this thesis, different system and development approaches such as BehaviourTree, AutoDesk Cognition, ABODE and Pogamut will be detailed in Section 2.3. Those systems and approaches can form the core or central parts for new game AI systems. Additionally, the design process should not limit the creativity of the designer to an extent where he or she is not able to express the desired behaviour of a character. Not having sufficient software tools at hand then sometimes results in paper prototypes or drawn sketches of what the game characters should perform. The actual process of designing game characters is highly similar to the process of writing stories for books and movies with the exception that the resulting character might interact in ways and situations which cannot be specified in a written document. Games are interactive media which allow the player to decide when and how to interact with objects. This exploratory freedom introduces dynamics which go beyond what

[4]Designer in this context are not visual artists Novak [2011, pp.319]. They are closer to writers in the movie industry than artists or programmers.

can be expressed in a linear written story. To illustrate the previous point, imagine following example within an adventure-type game:

> A player is always able to perform one out of three basic actions at any given time: *move*, *interact*, *doNothing*. The player can interact once per second. The game tries to respond to the state of the player, taking previous states into account to advance the story. After the first second, the player could be in one out of three possible states. Taking the previous state of the player into account, at the end of the next second, the player could be in any one out of nine possible states. After 10 seconds, the player could be in any one out of 3^{10} possible states.

This is an over-complicated example underestimating the power of game designers, as they would not track the progress in such a way. However, similar mechanics exist in role-playing games where the state is heavily dependent on previous actions resulting in an enormous amount of options to consider. In books, those options rarely exist making it easier to create a sound plot as the character is not controlled by the reader.

This process of designing game characters involves specifying all elements relating to the character and its interaction with either the environment or with other characters, including the player. Building on the earlier argument of Harbour [2004], there is no need to develop something if the player does not see or experience it. So complex social interactions between the NPCs are not required if the player is not around. To have an underlying model of those interactions which can be simulated and advanced without using their embodied representation might, however, be a better way of integrating a richer environment.

After the specification of the desired character behaviour, the programmers implement said behaviour, which in turn is evaluated by the designer. This process is incremental until the features and behaviours are either as intended by the designer or close enough to the design to move onto another feature. During the production phase, features are continuously added and altered by designers and programmers until the game reaches a state worth testing. The full feature set should be part of the design document. Though based on the development, new features can still be added during production.

Once the game is in alpha or beta stage, the game characters will be adjusted again. This adjustment is based on the user feedback and testing. During those phases, the whole game will be balanced and modified. This is an important step, both, the programmer and the designer are too close to the actual implementation to spot some of the inconsistencies or logical errors. Another purpose is also to move the game from

its current state closer to what the target audience—the players—find enjoyable. For the game character, it means that the designer is now able to see how players interact with their creation and if the plot of the game unfolds as anticipated.

1.3 Identifying Critical Points in Game Development

The previous section described the process involved in designing games and game characters and their position within the global game development phases. By analysing this process, a list of critical points in the process chain and in AI development becomes visible.

- Developing or selecting a suitable **game AI system** can create a critical situation for the whole game development because this decision is based on the personal experience of the team during the pre-production and on the initial design decisions. The problem when developing a new system from scratch is, the original project changes from developing within a given frame to first developing a new frame and then, as a second step, develop the game within it. This process takes extra time and resources and to arrive in the end at a stable game environment *and* develop the game in time is challenging. The game AI system is usually one of the most complex systems of a game in terms maximising the amount of expressiveness of the AI while maintaining the lowest possible computational footprint. It comes with its own problems which can, in the worst case, affect the entire game development. Choosing an existing system can also impact the game development due to unforeseen limitations or requirements to the system which come up late during the development.

- Providing ways for better inclusion of non-code based design as well as **visual editing or representation** of logical components in a robust way requires extra work. Due to the simplicity of state machines, which will be discussed in Section 2.1.1, they are the go-to solution for representation and creation of character behaviour [Rabin, 2010a]. Choosing existing systems such as AutoDesk Cognition, might also not always be suitable due to the dependencies with the underlying game AI system or because of licensing issues or the ability to extend the system in a desired way.

- The deciding on the correct **development process model** is a crucial task. A more rigid model such as the Waterfall is prone to problems when late requirements or problems need to be addressed in the development. Most game developers are transitioning now from classical models such as the Waterfall to

more agile methods for the development. Nonetheless, the process underlying the development still requires incremental milestones, based on the three basic phases. Even in more agile models, which respond better to changing requirements such as the model Keith proposed, unclear or uncertain requirements can produce delays in production as is expected. However, they handle changing requirements better due to their flexible nature.

- The inclusion of **player feedback** during the Alpha and Beta phases is a critical decision point during the development. It can create a large backlash of additional changes to the game. This process, often discrete instead of continuous, generates only data points for certain questions and problems of the development at testing intervals. This is due to the high amount of work needed to integrate testing into the existing process and due to the time it takes to evaluate the mostly qualitative feedback of testers. Smaller game projects cannot afford such extensive testing. Even for larger projects, the test phase is not a simple matter as it scales up in complexity due to a larger feature set to test.

- **Interdisciplinary collaboration** adds additional complexity to a project. Each discipline has its own language and approaches when tackling a problem making the coordination of efforts more complex. Games are no exception to that as the work between designers, programmers and artists is crucial for successfully completing features during development. Most of this work is tackled in a largely iterative way. The inclusion or alteration of artwork is generally less problematic as for most game engines fixed pipelines for artwork exist, making the process less dependent on others. For the inclusion of new game design features or game agents, the standard approach is mostly programmer-driven. In this case, the designer is heavily dependent on the programmer. This is based on the assumption that the programmer is including the game character specification and then the designer needs to validate the implemented behaviour [Snavely, 2004]. This can create bottlenecks during the development. For complex game characters, however, this continuous back and forth between designer and programmer can be quite time-consuming as most designers are not allowed to modify the actual game code and, as mentioned, require a programmer to integrate changes.

- The **design of game characters** is based to a major part on creative writing and storyboarding techniques [Stirling, 2010] borrowed from literature and film or movie productions. Those techniques were developed with a linear story and narrative in mind. Digital games, in contrast to written stories, offer a non-linear space for expressing stories and are more interactive allowing the player

to explore spaces outside a single designed narrative. This can draw out more creative potential compared to regular film or books, but it also creates more pressure on the designer to navigate a player through an experience.

- Another important point is the **designer toolbox**, a metaphor representing a set of techniques and tools, which can be used to enrich the behaviour of a game character. Normally, computational techniques are used by the programmer to include features requested by the designer. However, the state machine, for example, works well on paper or in a simple text-based form to specify certain behaviours which can be used by a non-programmer as well. Approaches like flocking behaviour from biology or higher level models of emotions and the appropriate response can be used by designers as well as "white-box" or "black-box" solutions, depending on the support for adding and editing the behaviour as a non-programmer.

- The overall **level of automation** is another potential area which is crucial. Most of the parameter tuning for game agents is either done manually by the programmer based on requests from the game designer, or automated and only active until the end of testing in the production phase. After finishing production, the automatic adjustments are either disabled or removed completely. However, some of the parameters are based on the testing phases and could benefit from additional adaption once the product is shipped [Brandy, 2010]. Those parameters are able to impact on the game experience, and it is crucial for a game developer not to create negative experiences for the player.

These critical points are decision points which can have a large effect on the overall production process. They are by no means intended to reflect wrong or bad decisions in the development process or the game development but are points where further work or research is possibly of benefit to game development.

The identification of critical elements was carried out by analysing the available literature on game production Thorn [2013]; Rabin [2010b]; Harbour [2004]; Novak [2011]; O'Donnell [2009]; Chandler [2009] and related online resources [Champandard, 2007c; Mark, 2012; Champandard, 2012]. The identified points in game development are not to be tackled in this work in their entirety. However, key elements will be addressed in the approach presented in the next section.

1.4 Motivation for Providing a New Approach to Agent Design

Understanding how human players approach games and how to guide their play has been one of the major questions ever since games were first created[5]. Interpreting the game's perceived performance and incrementally adapting them towards the player to create a better play experience is tedious and time-consuming. The process involves a tight iterative cycle occupying time for both designers and programmers as discussed by Snavely [2004]. Even for non-digital board games, a lot of play testing is needed to balance games. For digital games, this approach is even more complex as artificial players need to be included into the equation. Game designers now not only have to deal with balanced game mechanics that work, but also with the previously mentioned artificial players. Those artificial entities have a large impact on the player's experience because they dominate the player interaction with the game in most game genres. One of our fundamental assumptions of this work is that for better artificial players the human players themselves need to be understood or modelled better and this is where the presented work expands existing work on generative agent creation.

Based on the previous section, a number of elements were identified and highlighted out of them a selected subset will be addressed in this work. The subset was selected based on its believed merit for novice developers, game designers or developers. This target group does not use or rely on programming and can be supported with a robust, scalable approach and an adjusted sound work-flow taking into account interdisciplinary teamwork.

One of the key elements in this work is to support the creative input of users into the design and development process. Creative in this context does not refer to purely artistic expressions [Kelley and Kelley, 2013] but also includes human input in the form of strategies, plans and exhibited behaviour. Thus, artefacts which are not specified purely as program code or abstract numerical values. To take forward an argument made by Snavely [2004] that it is possible to empower designers and integrate them better in the overall work-flow of game development. He presents an approach using Microsoft's Excel spreadsheet tool in combination with a set of Visual Basic scripts to allow designers to specify character attributes and parameters in tables. This is based on his experience and observations and on the assumption that for statistical information or attributes designers already employ an approach which can be presented through fuzzy set theory (FST) [Zimmermann, 2001], an assumption supported by the

[5]Analysing human play even in digital games is mostly based on early works on child's play by Huizinga [1950].

work of Zimmermann [2001]. Fuzzy set theory was intended to support modelling, especially in cases where explicit criteria are not present. Thus, a vague specification is available, but a precise answer needs to be present in the end. An example should illustrate this case:

> A designer wants to specify the age of different characters roaming a city. Instead of using precise numbers for the ages of each individual he or she has specified five young and ten middle-aged people. However, this definition in natural language when it comes to age is a fuzzy definition with soft boundaries between the two given age groups. It is easy to say that somebody who is above 30 years fits into the middle-aged category or that somebody who is younger than 20 years is young. It is, however, less easy to categorise people around their 20s.

FST provides for those areas a membership function which offers a way to map individuals with a certain percentage into the correct group by assigning a value from $[0, 1]$ to the describing attribute and how precisely they match. This allows the usage of vague linguistic descriptions to be used in a precise manner.

This notion resonates well with initial states in the game development process where only rough ideas are known about the criteria of a specific agent. For sport, management or role-playing games attributes and parameters are essential to gameplay as they can easily be mapped into values such as strength, or income which makes the transition from set theory easier. However, according to Snavely, designers are usually only specifying lists of those attributes which are then included at a later point by a programmer. He introduces spreadsheets as tools useful for design. By using a spreadsheet he argues, designers are able to export data in a standardised, stable format and structure which creates a new pipeline[6]. This renders the work-flow of designing and implementing agents from programmer-driven to designer-driven. The data structure, which the designer can export, can be imported into a game without creating problems due to a "wrong format" or missing information as this can easily be handled by the export tool from a spreadsheet.

This supports one of the assumptions of this work, that including designer input in the forms of imprecise strategies or plans instead of purely mathematical descriptions will provide a more direct way for designers to interact with the game. The result should be the removal of tight bottlenecks when game or level designers need to involve programmers to test their changes in an iterative fashion. Extending now the metaphor

[6] A pipeline in this context refers to a stable process where the input is specified, and a defined outcome is generated.

of game designers to other groups we can include novice programmers as well into the scope of our approach.

In this thesis, we take the argument a little bit further, extending it into areas where it is not just statistical data that can be represented in a spreadsheet. As observed by O'Donnell [2012], an approach is needed that redistributes the workload and dependencies between the multi-disciplinary team members more evenly. This envisioned approach needs to take into account that there should also be a focus on components which allow the inclusion and alteration of relevant game information in a robust and secure manner to—at the same time—not allow for the alteration of the underlying program code. A specific interface and approach including the designers is needed.

1.5 A New Integrated Process of Agent Design

Motivated by the need for a less programmer-driven approach to agent design in games, in this thesis, I present an approach which focuses on a clear pipeline between designer and programmer. This new process model addresses the need for such an approach discussed by O'Donnell [2012] without ignoring the software aspect of game development. In contrast to Snavely [2004], who uses spreadsheets, a more general approach and methodology to design is needed. Separate tool-chains are advantageous as these tool-chains create less coupling between software components supporting the robustness and stability of the process similar to the benefit of loose coupling in Software Design.

Bryson and Stein [2001] introduce an approach for modelling behaviour-based AI in an iterative fashion, illustrated in more detail in Chapter 2.2.9. Thereby, they allow for iterative changes to the agent, increasing the capabilities of its intended behaviour in stages. This approach—Behaviour-Oriented Design (BOD)—aligns to a large extent with the current game development methodology in games. This thesis extends the given approach and focuses heavily on the creation of a robust and user-centred workflow for designer and programmers including new tools to support the design and development process. The work integrates those additional tools to allow a simple way to create more flexible and non-deterministic behaviours using a bio-mimetic plugin. The approach fosters the idea of using evolutionary methods, as demonstrated in Chapter 6.6, as part of the design process to allow for a better exploitation of the behaviour space, which follows the argument of Champandard [2012].

1.5.1 Research Question

How can the design cost for behaviour-based AI be reduced?

Based on this research question, different sub-questions emerge which, if combined, address the overarching question of design cost reduction for behavioural AI in games and related areas that require designable agents for specific experiences such as entertainment or education.

Sub-questions:

RQ1 Are there substantial commonalities between agent architectures?

RQ2 What are the requirements for bridging the gap between academic and industrial games research and agent design?

RQ3 Is it feasible to integrate non-programming knowledge into the design of agents?

RQ4 How can a lay user be supported in designing complex game AI?

RQ5 Human learning is partially based on mimicking—a powerful technique to observe and reproduce. Can this principle be applied to aid agent design?

1.5.2 Contributions

- *Agile Behaviour Design* is a new methodology for game AI development which advances BOD by introducing agile techniques from software development into a more directed and supporting approach. The new methodology was evaluated using a new light-weight architecture and supports task-sharing and teamwork by a stronger separation of individual skills and responsibilities. Furthermore, it strengthens the scalability of BOD to more complex agents.

- POSH-SHARP, new agent framework for creating agents for multiple platforms such as mobile phones or web applications has been developed and evaluated. The new system, discussed in Chapter 5, allows the development of light-weight game AI systems and includes extensions such *behaviour versioning* and *behaviour inspection* to increase the robustness of new agent systems and support the development of game AI for novice users.The systems addresses a subset of the identified weaknesses of all surveyed architectures and provides a novel platform for experimentation.

- ERGO is a new general-purpose, low-cost mechanism for altering the selection process of goals and behaviours. The new augmentation of behaviour-based AI

systems such as IVAs introduces a new form of memory into the selection process. This summatory memory allows a behaviour to take control based on an internal motivational state or need. ERGo is a new light-weight bio-mimetic mechanism for creating non-deterministic behaviour that can be designed and addresses the industrial need for easy to integrate but flexible approaches to selection goals.The approach is presented in Chapter 6.

- A survey of the state of the art of game AI techniques is presented in Chapter 2. This survey integrates academic and industrial research on components, architectures and approaches to IVA into a wholistic view on the topic. The survey also illustrates how game AI shifted over time not only driven by hardware development but also by the availability of new techniques. The result of the survey was crucial for the identification of weaknesses of current game AI systems and highlighted the importance of developing a new software process.

- A new genetic programming approach based on recorded input data from human players is presented. This approach evolves artificial players in a form amendable for further authoring. This is the first time that an approach offers a "learning by example"-way of evolving non-trivial, understandable and amendable agents as executable program code. In contrast to other approaches, designers are able to play a game, record their interaction and use the approach from Chapter 7 to create agents which can be manually enhanced and edited later on. The developed approach evolves artificial players using PLAY TRACEs in the form of Java program code allows for the inspection of possible underlying models of the player's motivation or reasoning process. The approach is presented in Chapter 7.

In the final section of this chapter we discuss how this approach was developed and the story underlying this thesis. It focuses on a time-line discussion of the research carried out and links to later chapters. The approach itself will be extended and incrementally built up during this work, resulting in a final presentation and discussion in Chapter 8.

1.6 Overview of the Thesis

The thesis is organised in the order in which I structured the approach to analysing and advancing the design of game AI systems for multi-platform development and application. The work starts with an initial survey of existing agent architectures and approaches for designing game AI, which is covered in Chapter 2.

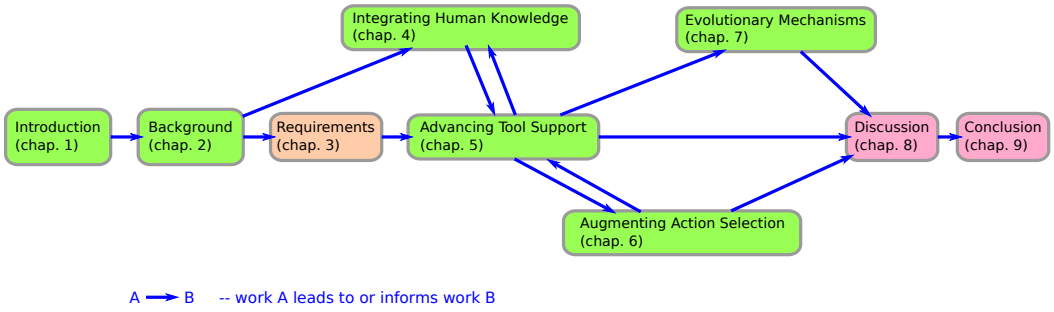


Figure 1-3: The chapter overview ilustrates the organisation of the thesis. The chapters are organised based on their chronological order; earlier chapters motivate later chapters. Chapter 4 and chapter 5 are however in a cyclic relationship as the first part of chapter motivates chapter 5 and the later part of chapter 4 builds upon chapter 5. A similar relationship exists between chapter 5 and chapter 6 as chapter extends chapter 5 and presents a crucial part of POSH-SHARP.

This is followed in Chapter 3 by an analysis of three IVAs architectures through repeated interviews with development teams for those architectures. During those interviews important steps uniting and differentiating the three approaches are identified and discussed and general problems with the design of IVAs are collected.

In Chapter 4, a genuine industrial game AI environment, StarCraft in combination with [BWAPI Development Team, 2010], was chosen for experimenting and deepening the understanding of commercial game AI systems. StarCraft offers varied opportunities to research different aspects of artificial or computational intelligence ranging from planning problems [Soemers, 2014; Weber et al., 2010b] and Chapter 4.3 to optimisation problems [Weber et al., 2011; Justesen et al., 2014; Shantia et al., 2011]. It is frequently featured at major game AI meetings.

Two specific directions are at the centre of focus while carrying out this research:

- The first is how to enhance the planning process to make the development more robust while maintaining a clearly structured approach that is able to scale well to different problems and team sizes.
- The second direction is focused on how to enhance
  (a) the design of the resulting agents and
  (b) to support the planning process of a modular approach during the development of an agent.

During this initial phase, the AI architecture jyPOSH is used—a POSH [Bryson, 2000b] version written in jython[7]. As a starting point for the design, Behaviour-oriented Design (BOD) by Bryson [2001] is chosen. During this phase, the main objectives are to observe how the design of a complex game AI is approached and in which way the methodology needs to be altered to better fit the process.

In Chapter 5, potential elements for advancements in the chosen methodology and architecture were identified during reviewing the state of the art in the related literature. These changes are supported by observations made during the development of the first prototypical AI for StarCraft [Gaudl et al., 2013] as well as the game agent development undertaken as part of the Bath Computer Science course "Intelligent Control and Cognitive Systems". A new version of POSH—POSH-SHARP—addressing a first set of discovered shortcomings is proposed as a first result. The proposed changes take the previously mentioned research directions of supporting both the planning and the design into account integrating knowledge from software design and software engineering principles. During this phase, a streamlined design of agents is developed applying software code annotations and code inspection to the new library. Additionally, enhancements to the iterative development process are proposed to include a more robust transition between different versions of the agent code. Alterations to the deployment of the resulting agent and library will be described which aim to reduce the effect of user errors by switching to a more robust library distribution format.

Chapter 6 describes an analysis of the underlying behaviour arbitration process employed by our planner. The arbitration process is what is responsible for selecting an action the agent should execute, given a corresponding set of sensory inputs. As a part of this research, the extended ramp goal model (ERGo) was developed. ERGo

[7] JYTHON integrates both python and java into a new OOD hybrid. It allows the usage of packages from both roots. This combines the advantages and, more importantly, the disadvantages of the individuals root languages.

continues the idea of a flexible latch when arbitrating between behaviours of equal importance [Rohlfshagen and Bryson, 2010]. ERGo in contrast to the Latch directly alters the selection process resulting in a different arbitration process. This new mechanism is based on phenomena observable in the mammalian brain when maintaining or inhibiting behaviours.

Additionally, a mobile game is presented which contains four cognitive agents interacting with the player. Those agents are augmented with the newly proposed concept—ERGo, demonstrating the feasibility and performance of said approach in actual practice. The developed game won the IEEE CIS mobile app award in 2014, see `http://cis-mobileapp.deib.polimi.it/plcis/index.php`. To build up on the previous changes, the research is carried out using the new version of the POSH planner– POSH-SHARP– which extends the previous version.

In Chapter 7, after discussing alterations to the design process and methodology, extensions to the reactive planner and the impact of fully automatic generation of reactive plans are more closely investigated. This includes the potential impact on the design process. In this part, generative systems and approaches are discussed with a focus on fully autonomous systems versus fully hand-authored design.

By using genetic programming and recorded human play, a new middle way between those two extremes is proposed. This new approach creates artificial agents by mimicking human play. The created agents learn to behave like a particular natural agent from the provided data. This is demonstrated using the platformersAI toolkit which offers an environment similar to commercial two-dimensional platform games like SuperMario from Nintendo.

Finally, in Chapter 8, the findings are summed up and contributions of this thesis are discussed together highlighting the advancements of new design methodology. Thereby, we discuss possible next steps and open research directions which can result from continuing this work in specific directions.

The next chapter is a background analysis on existing game development techniques and agent frameworks. It discusses and compares existing approaches and identifies weaknesses in industrial and academic approaches to agent design. Those weaknesses are essential for advancing agent design and derive results which could be utilised by the games industry and academics alike.

Chapter 2

Background

This chapter provides the survey of the current state of the art and is intended to be used as a common ground and a foundation for concepts and methodologies introduced in this work and which are relevant to game development in general. This is even more important as this thesis aims to connect academic understanding and concepts to industrial applications and conventions. Due to the differences in academia and industry and the fast evolution of the games industry itself some terms are often used loosely or interpreted differently on either side of this divide.

The chapter is also meant as a primer. It gives insights and starting points and background literature needed for working on or researching game AI. It illustrates the current state of the art of AI at the time of writing. This is done all under the special lens of applicability to games.

The chapter is divided into four sections which emphasise different aspects of game AI design.

- Section 2.1 introduces fundamental techniques used in games such as decision-making, spatial-reasoning and evolutionary approaches. They will be referenced throughout the thesis and are necessary for understanding the underlying frameworks of all industrial game AI systems.

- Section 2.2 introduces the concept of agents and discusses higher level approaches for designing them. Having a clear definition of agents is essential to understanding the difficulties of designing them. This section also includes discussions on three dominant cognitive architectures which have been used in game related research.

- Section 2.3 goes beyond concepts and approaches and presents and discusses existing software tools and of those can support developers and designers when building agents.

- The final section of the chapter summarises the findings of the survey and the state of the art of Interactive Virtual Agent (IVA) design.

This thesis focuses on a specific domain, digital games, thus, it is more natural to illustrate most of the cases with examples close or related to games. Digital games are computational systems so most of the approaches can also be employed in other domains such as robotics. Digital games similar to robotic systems have individual, unique characteristics making them on one side highly interesting on the other side hard to work with.

Commercial games typically have strict resource limitations. Those depend on the current hardware platform and the scale of the developed game. This means that certain techniques are not feasible at some point or, that they made their debut into practice relatively late in contrast to academic AI. In the field of robotics, issues are mostly related to mechanics or uncertainty in sensing because robotic research is either heavily centred around solving mechanical or engineering problems robots face when interacting with a physical environment. Another reason is that even solving lower level problems is extremely hard due to the high dimensionality of information available in the physical world. This indicates that it is currently difficult to implement higher level reasoning on operating a car when it already requires most of the resources of a robot—or a car in that case—to identify and approach physical objects such as a road or a person by using real-time image feeds from the robots visual sensors.

When it comes to digital games and AI, one aspect of games is often overlooked. Digital games are designed to entertain the user in one way or the other. Thus, the goal of good game AI should not be to win against the users but to keep them challenged and entertained. This point is not only of great importance to the commercial product but brings in a separate degree of complexity into the AI and is often underrated and understated in research. It is easy to imagine games where the user is always losing to a superior AI, either because a winning strategy is known and employed by the AI system or because the AI simply is faster at finding the correct move to make at each given time. It can be argued that for complex games such as chess or GO[1] a winning strategy is not known and the search space simply is too big [Müller, 2002; Clark and Storkey, 2014]. For chess the number of possible moves is around 50 at each discrete

[1]For GO and board of sizes up to 5×5 the game can be solved [van der Werf et al., 2003]. Thus, a winning strategy is known, but currently not for boards larger than 7×7. This still holds, even though artificial players such as AlphaGO are able to beat expert players [Kasparov, 2016].

Systems that think like a human (C1).	Systems that think rationally. (C2)
Systems that act like a human. (C3)	Systems that act rationally. (C4)

Figure 2-1: The four areas of systems AI, which represent clusters of approaches as defined by Russell and Norvig [1995] .

time step whereas for Go the number of possible moves initially is 19×19—361 possible moves. Thus, we need to concentrate on search and optimisation strategies to provide satisfactory AI systems for games.

However, if we solely focus on advancing search strategies we miss an important previously mentioned point—games need to be entertaining for different groups of players as well. It is not enough to provide a highly challenging AI that is able to win against proficient players. We also need to address the scalability of skill to different player skills and the modelling aspect of desired behaviour. The last argument is one this thesis emphasises and works hand in hand with designed skill levels of users. It will be continued later on when discussing agent design after an introduction to the spectrum of what in this work is united under the term game AI.

2.1 Fundamental Game AI Techniques

To understand what game AI is, a good starting point is to think first about the broader area of artificial intelligence. Starting with the standard textbook definition of AI that was introduced by Russell and Norvig [1995], most AI approaches can be grouped into two main dimensions. In their definition, the first dimension is defined by reasoning versus action. The second is rationality versus human-likeness[2]. From those two dimensions they categorise approaches towards artificial intelligence into the four resulting sectors:

Approaches focusing on reasoning are found under the first row, C1 & C2. Those approaches deal with the non-visible part of a system's behaviour which is hidden to the observer. The difference between the two columns being that a rational picks the best possible solution to a problem out of a pool of given options. In contrast, not all human decision making is always rational. Human decision making also encompasses emotionally driven decisions, handling limited information [Simon, 1972] and self-deception [Rauwolf et al., 2015] which divert from rational decisions.

The second row in Figure 2-1 describes the exhibited parts of a behaviour, they are the generated actions visible to an observer. The important difference is, that

[2]In most cases human-likeness can be replaced with animal-likeness because not all that the field of AI tries to solve or understand is human reasoning alone.

the approaches C3 & C4 do not try to model the internal state or reproduce the exact reasoning process behind a given action but care about creating the appropriate or desired output. Approach C3 represent the group of systems which act like a human. Thus, the system is providing actions imitating human behaviour. Rationally acting systems, however, are not meant to imitate other entities. They act to achieve the best possible outcome.

Yannakakis [2012] discusses on a high level the history of academic game AI and related research. This provides a first starting point for identifying fields of interest in game AI research. However, for this work the presented overview is not detailed enough and is one-sided, as it only discusses the academic perspective not taking industrial publications into account.

The remainder of this chapter does not try to repeat or duplicate the effort undertaken in creating an all-encompassing collection or a high-level overview but to describe a subset of techniques mentioned by Russell and Norvig [1995] and Yannakakis [2012]. It contains a subset of the current state of the art techniques found and used in digital games integrating industrial and academic knowledge. This is done to aid the understanding of this work and its context. Additionally, it is important to note that the approaches from the game industry are mostly favoured by its developers due to the strong focus on applicability, ease of implementation and efficiency described in game specific publications. Prominent publication series are the AI Game Programming Wisdom and the Game Programming Gems series which are long running and peer-reviewed by professionals.

The following sections first present an overview of decision modelling approaches in games. Those approaches are ordered by their first appearance in games to illustrate how game developers adapted to the changed requirements. This is followed by a discussion of more computational intense approach for spatially centred search. Spatially centred search is a large part of game AI and also aids the design process when generating games. After discussing spatial approaches we move to the most expensive techniques used in games which integrate learning or adaption during the execution of the game into the software. After discussing what current fundamental game AI techniques are, we will have a closer look at agents and agent design used in games which cover higher level approaches that can incorporate those fundamentals.

2.1.1 Decision Modelling

Ad hoc Rule-Statements

The simplest and most common technique used in games is a rule statement also called an ad hoc rule-based system. This system is used to create basic reactive AI systems for games. Instead of using abstract concepts such as the later discussed finite state machine or systems which can be generalised, this approach uses rules specified in the programming language of the used system throughout the actual code base. An example of such rule statements can be found in Figure 2-2. The rules tell the system, based on current input and memory, what to do. They are usually implemented as `if-else`[3] or `case` statements. Ad hoc rules differ fundamentally from rule-based systems used in academia. They are basic chained logic statements and are not interpreted or used with an inference engine as in expert systems but executed at run-time. The rule statements are also not linked to each other which means they are distributed throughout the whole game and only react when appropriate signals occur. Due to their simple structure, they are hard to maintain once the logic gets too complex. Problems can occur when using either multiple nested statements or when the logical statements contradict each other. As there is no higher level control over them, contradictions or error in the logic have to be checked manually.

They are initially easy to implement but brittle when adjusted and maintained for a longer period of time. However, for smaller systems or rapid initial prototyping they are perfect. Prototypes initially are quite limited and the rules are easy to implement, making them a good pair. After a prototype, most of the code will not be used in the later systems so scalability or maintenance issues are not an issue for them. For small systems, debugging ad hoc rules is simple as we only need to follow the execution queue of statements. As described earlier, they are logical statements and are directly embedded in the code. This makes them nearly unusable for designers in the actual game as they are part of the core system and designer generally have no access to those parts of a game. In chapter 5.3 a new decision making framework is presented which addresses this issue of having a simple design approach while separating the underlying game architecture from the design.

Message-Based Systems

The second approach commonly used is the message system. This system communicates between different game objects using a messaging infrastructure. The approach is

[3]Throughout this work inline code will be presented using `this font` to allow a correlation between text and example code.

```
if (playerVisible){
  if (botHeath > 40.0f){
    attack();
  } else {
  flee();
  }
} else {
  explore();
}
```

Figure 2-2: A simple set of ad hoc rules encoded as nested if-else statements. Code blocks like this are useful for controlling a bot in a simple game.

relatively fast to implement by following a standard software design pattern for a listener structure. Message systems scale well to larger systems. However, due to the asynchronous sending of messages, it is hard to follow the signal flow or to predict which game objects interact with each other. For larger systems it is even difficult to follow the execution chain, making them quite hard to debug. Message systems are powerful in terms of distributing information between different objects. They allow a component to broadcast a message to multiple other components without directly addressing them. However, due to the distributed communication, the overhead can be quite dramatic. They are also harder to iteratively develop and maintain as a change in protocol requires changes for all Listeners. One of the significant downsides of message-based systems is that they are again purely programmer-driven and hard to understand and design by a non-programmer. There is also no easy way to visualise or edit message-based systems.

Finite State Machines (FSM)

One of the easiest and still widely used standard techniques is the Finite-State Machine (FSM). Games such as Namco's PacMan for Arcade cabinets [Montfort and Bogost, 2009] utilised the concept of state transitions and conceptualised states in design as FSMs [Bourg and Seemann, 2004, p. 165]. However, PacMan was implemented using tile systems[4] which contained all of the actual logic instead of using an implementation of states and transitions. Due to the usage of lower level languages and the restrictions on memory, advanced approaches or complex control structures were not possible. Thus, the game architecture was not differentiating between logic for characters and

[4]A tile system uses the underlying world representation to contain the control logic. Tile systems segment the world into fixed sized chunks, normally squares. Instead of having a centralised logic system which has to memorise world positions, each tile contains a part of the logic allowing the game to use local reasoning around a tile rather than global reasoning. This approach is highly memory efficient and works well for local reasoning.

environment or rendering text on the screen. The original PacMan logic that controlled the entities in the game—the ghosts—was bound to the underlying tiles of the level—the before mentioned tile system— and not to individual agents. The process implementing the game was completely done by a programmer, whereas the initial design was done on paper.

FSMs offer an intuitive way[5] for modelling behaviour by using states and transitions between them. They can be designed on paper by non-programmers and offer great design freedom as they decouple the design and the implementation of control logic. Meaning, the designer can layout the intended behaviour of a system initially without writing a single line of program code. Later on, a programmer can translate the resulting graph into program code. Figure 2-3 is illustrating an FSM for one of the entities controlled by the PAC-Man game.

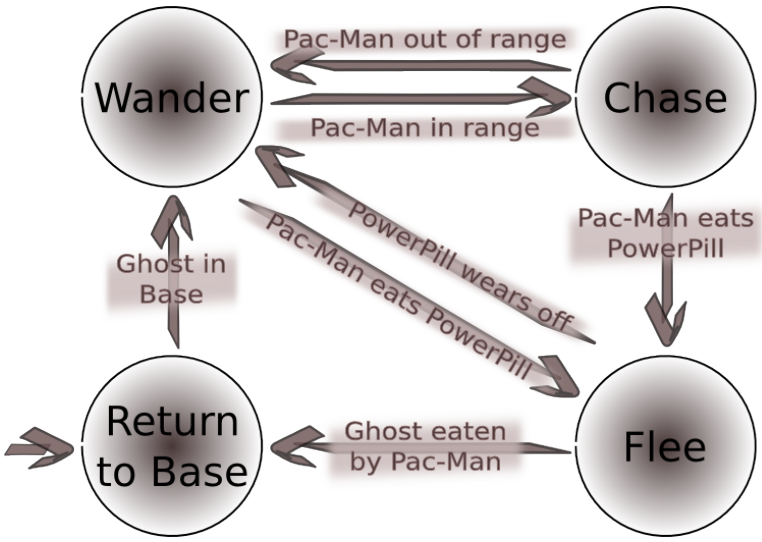


Figure 2-3: A simple finite state machine for one of the four agents in Pac-Man. The FSM contains four states: "Return To Base", "Wander", "Chase" and "Flee". The start state is "Return To Base" and there are accepting states as the state machine should loop through all states until the level is done.

FSMs can be visually represented as directed graphs that can additionally contain entry or starting states and accepting/holding states. The latter define valid states for exiting the FSM. Thus, the state machine always starts in the entry state and is progressing through all states and until it stops correctly. An FSM stops correctly when a transition ends on an accepting state. Due to this graphical approach to structuring logic, they offer an excellent way to sketch out behaviour for characters. This behaviour design using

[5]The term intuitive will be used throughout this thesis to describe ease of design and a shallow learning curve as described by Raskin [1994].

states and transitions can as well be done on paper. A state only changes from one to another by using a transition, a directed edge linking two states. In the Pac-Man state machine, see Figure 2-3, Return to Base is the entry state into the FSM. For the designed Pac-Man entity—a Ghost—there are no accepting states because the entities start and stop with the game which leads to a design that does not require a valid end state for a ghost. The game ends either when the player dies or when a new level is started.

initial state	r	-r	p	-p	g	e
s1	-	-	-	-	s2	-
s2	s3	-	s4	-	-	-
s3	-	s1	s4	-	-	-
s4	-	-	-	s2	-	s1

Figure 2-4: A transition table for Pac-Man based on the ghost behaviour of Figure 2-3. The transition criteria are: r (Pac-Man in range), p (Pac-Man has Powerpill), g (ghost in Base) and e (Ghost eaten). The states used are: s1 (Return to Base), s2 (Wander), s3 (Chase) and s4 (Flee).

Another common representation form for FSMs are transition tables, see Figure 2-4. They define transitions from one state to another in a table where each row is setting transitions for a specific state to another. The larger a state machine gets, the easier it is to use a transition table. It is additionally beneficial to see missing transitions as you can check the particular row associated with a state. Once a new transition is added, each row in the transition table needs to be checked and adjusted. This checking procedure is a simple task in a small state machine such as the one presented here.

However, both representations—the graph and the transition table—suffer from similar problems. For large systems, it becomes hard to visualise or grasp the graphs or tables. Maintenance and modification of an FSM are hard as each transition from one state to another needs to be checked and updated whenever the intended logic changes. As one state can have multiple transitions entering or leaving them, all connected transitions need to be updated. So, large late changes of an existing graph are expensive and prone to produce errors or result in different behaviour. Another issue with state machine is that it often is encoded as switch statements or if−else blocks in the actual software. This hard-coding of a decision structure means that it is hard to check the final realisation for correctness regarding the difference to the written down specification, at least for a sufficiently complex system.

FSMs are frequently embedded as program code instead of using an external structure. A designer, in most of those cases, is only specifying the intended behaviour on paper which is later on implemented by a programmer. This stages process creates a fragile and complex dependency between programmer and designer for including later

changes or testing; it involves two people for one task. Even though FSMs are easy to implement and visually check for smaller graphs, it is hard to verify or validate large resulting systems as they mostly resemble non-deterministic Turing Machines and require more work than simple parameter sweeps.

Due to the growing complexity of the tasks the system has to perform and react to, hierarchies were introduced into the state machines. This additional level of abstraction allows for a better modularity of the system. It also made the systems harder to design on paper as it involves now having nested state machines, meaning multiple graphs. Brooks [1986] decided to introduce a different technique into his reactive system and arranged conventional state machines in an layered order in his SUBSUMPTION architecture.

The transition table would now contain links to different entry states in other state machines and there would always be a need for specifying valid end states for a nested state machine to trace problems.

Due to their relatively shallow learning curve and their expressive representations, FSMs are even now a popular technique in current commercial game systems. An additional bonus is that they are fast to implement, in terms of required programming, and their additional computational overhead is extremely low. Thus, it is possible to run and maintain a large set of state machines even under severe resource restrictions. FSMs have been used as part of game AI systems since some of the earliest games and are still used in commercial games such as BATMAN ARKHAM ASYLUM by ROCKSTEADY STUDIOS [Thompson, 2014]. FSMs are suited for small, purely reactive systems, or ideally subsystems in a larger AI system. They do not support planning, prediction or memory but offer an intuitive logic representation. The AI approach for Batman utilised the easy creation and low overhead of state machines to rapidly prototype specialised behaviour for their game characters [Thompson, 2014]. They designed a new feature and implemented it normally in under a day. Due to the small size of those FSMs, they are easy to tweak and adjust for a programmer. As each of the features was to a large part independent of the rest, this approach allowed for parallel work on features and iterative development. However, the resulting system is then heavily dependent on the maturity and robustness of each feature.

Utility-based Modelling

This variation of game AI approaches is based on economics and game theory. The basic assumption is to optimise a set of behaviours by fitting them to a particular function or curve, instead of modelling behaviour based on human design. Thus, the goal is to have behaviours which maximise their utility. The approach to model elements of

game AI based on optimising certain parameters is not new but according to Mark [2009], it offers more flexibility to the system and is modular and extensible in contrast to previous approaches. For games where the performance of an AI can be numerically evaluated like simulation or management games, this approach fits perfectly. However, the approach also works for other games such as role-playing games. In those games the system tries to optimise internal parameters like health in combat situations or fame when it comes to more global parameters.

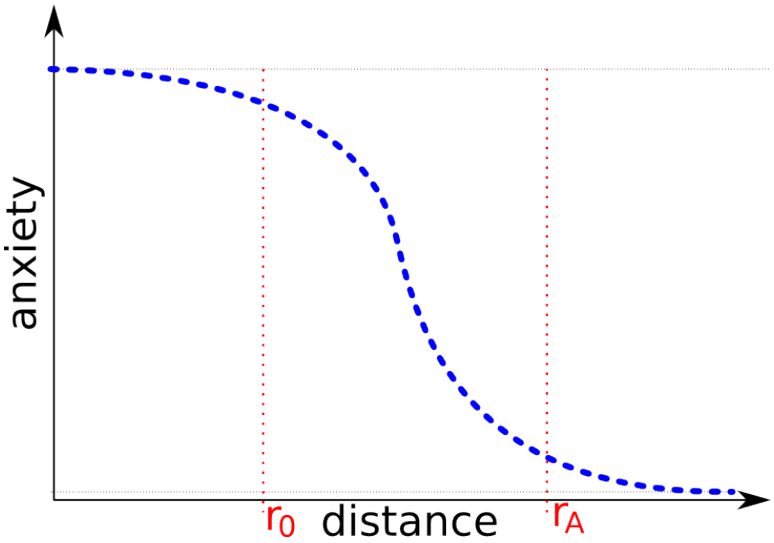


Figure 2-5: A sigmoid function used to determine when to release a trigger. At distance r_A the target is in range. At distance r_0 it is safe to assume that we cannot miss the target any more.

The approach, due to its mathematical underpinnings, can be used in large-scale systems and translates to different disciplines as well. The evaluation of a function is generally cheap compared to other approaches. As this approach relies on optimising towards a certain function, it needs to fit the intended, known criteria. Thus, a linear mapping between an interesting function and parameters of the system needs to be known beforehand. To illustrate utility-based modelling for game AI imagine following setting based on a talk by Mark [2010]:

> Player A has a water pistol which has a useful range of r_A. Beyond that range, the player cannot hit a target. At range r_0 however, it is safe to assume that the player will definitely hit the target with maximum effect. Player B moves along a path getting him closer to Player A. Once player B enters the range of player B, she could potentially shoot the water gun. However, player B could instantly step out of the range resulting in only

> dealing minimal damage, wet shoes. To maximise the damage and create an "interesting" player behaviour player A could utilise a sigmoid function, or "Logistic Function" [Mark, 2010] see Figure 2-5, to determine when to shoot the water gun. The function and the usage of the two thresholds creates a more dynamic way to release the trigger, yet maximised amount of water damage without specifying a fixed range parameter for when to release.

Utility modelling provides a low learning curve for programmers because it comes down to implementing given mathematical formulas, and a low complexity in modelling decisions which can be numerically evaluated. The technique is quite popular in early phases of game development. Schmidt [2015] use a utility-based system for modelling their NPCs' decision process. A main motivation underlying their design decision is that different enemies should feel distinctly different when playing against them. They give the following example for their utility approach:

> A skeleton can perform three actions when engaging with a player unit:
>
> - The skeleton can "Chop"—a melee action to injure the opponent. If there is a low chance of landing a hit the utility to perform that action is low. [utility 3]
> - It can "Split Shield"—a melee action to destroy the opponent's shield. If the opponent has no shield, there is no need to perform that action and its utility is zero. If the opponent has a shield and it can be destroyed using the current weapon, the utility is high. [utility 9]
> - The monster can "Shieldwall" to protect against the opponent's attack. This action receives an average utility to guard the character. [utility 5]
>
> To decide now which action to take, the utility of each separate action is calculated based on an internal mechanism, independent of other actions. As a final step, the action with the highest utility is chosen. In our example, the skeleton would take the "Split Shield" action—as it has the highest utility—to destroy the shield of the player.

The advantage of this approach over a decision tree, ad-hoc rules or a finite state machine is that each action and its related utility is separate from others. Thus, new actions and their calculation functions can be added or old ones removed without destabilising the AI system itself. A fundamental issue, however, as discussed by Schmidt

[2015] in his blog entry is balancing or "designing" of perceived behaviours. This process is time-consuming. The initial approach is to select a function representing an idea of a certain utility. After implementation and testing, the function needs to be adjusted to address the difference between the actually exhibited behaviour and the intended one.

Taking available functions from economics and those illustrated by Mark [2010] into account, the outcome of the optimisation still needs to be as intended by the game designer. This is one of the biggest issues with this approach. Using a utility-based system to design AI requires knowing and identifying appropriate functions for different scenarios. Additionally, the approach relies to a large extent on the programmer. Leaving the designer only to suggest functions but not being able to integrate or test them as they need to be integrated by the programmer and tested while with both team members are present.

Game BehaviourTree (BT)

There are currently many definitions relating to the term BehaviorTree (Bt). The two most dominant definitions of the term are those of Isla [2005]; Champandard and Dunstan [2013] and Dromey [2003]. The first definition deals with the design of behaviour-based agents for game AI and originates from the games industry. The second definition dealing with software engineering and design requirements will be discussed in the next section.

In 2002 BehaviorTree was first used by Isla in the game Halo [Isla, 2005] and later advanced by Champandard [Champandard, 2003, 2007d,a; Champandard and Dunstan, 2013]. Having previously worked on c4 [Isla et al., 2001],which will be described later on, Isla moved into commercial game development. He initially described BehaviorTree as a hierarchical FSM or more specifically as a directed acyclic graph (DAG) for game behaviours. For the initial draft of what is now called a first generation BehaviourTree, he implied four principles: customisation, explicitness, hackability and iterative development. Those reflect the focus of what was required to employ the method into the Halo game. A first generation BT is presented in Figure 2-6.

Customisation reflects the concept of adjusting the decision-making process based on the desired output. In his initial design of the DAG and as a fallback the leaf nodes executed actions whereas the non-leaf nodes use customised code to make a decision regarding which subtree to follow. He proposes a two-step model. As a first step, the parent makes a decision on which child to execute. As a second step, the children can alter the outcome of the result by competing for execution. As this deliberation process of selecting a subtree is quite expensive and, games require tight control over the

resources, Isla modified the original design for the decision making by moving further away from a simple graph structure.

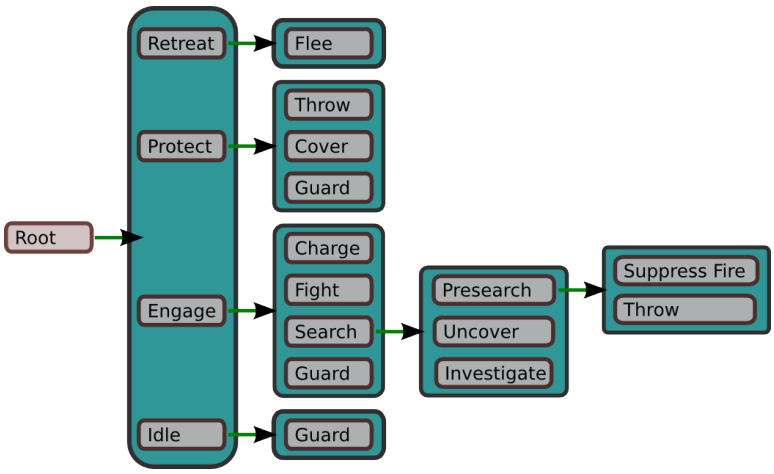


Figure 2-6: A directed acyclic graph (DAG) representing a first generation BehaviourTree (BT). The BT is based on Isla [2005] but does not contain BehaviourTags,Impulses or Stimulus Behaviours.

The new decision process also moved closer to the later described agent architectures including the c4 system [Isla et al., 2001], except that it still focuses heavily on maintaining a very low CPU load. As not all nodes are always relevant to the current execution of the tree, because some internal condition is not met, they are marked not relevant in the current cycle. This process can majorly impact the run-time of a game as it involved each node along the path through the graph. The relevance function can contain elements such as the status of an object that is useful to the given behaviour. An example would be that there is no need to use the sub-trees for driving the vehicle when the player is in the driver's seat and able to drive herself. The previously described conditions specified the foundation for an initial description of Bt. After this first version, the approach was altered to integrate the following node elements as an advancement:

- **priority-list:** A priority list of children which are executed highest priority first. If a child cannot execute, the next one in the list tries to execute until one can execute. For subsequent ticks, higher priority children can interrupt a running lower priority one.
- **sequential:** The children are visited in sequential order until the last child is visited and then the parent node is finished. Only relevant nodes are checked.
- **sequential-loop:** The children are visited in sequential order, same as above.

However, once the last element is reached the list is looped again and the parent is not finished.

- **probabilistic:** Picks a random relevant child that is relevant in the current context.
- **one-off:** Picks a child randomly or uses its priority order but never picks the same child twice.

The most favoured element in this list, the priority list, presented in talks at the AIGameDev conference[6], was kept unchanged in later versions of BT as well. By having a closer look, option four and five are relatively vague in their description. If child nodes have a probability of success assigned, the BT needs to track those changes and alter them. For large trees and frequent updates this process is costly. The *one-off* type reduces the number of children available for execution, resulting potentially in an empty list at some point. If this situation occurs at a vital time during the interaction with a player, the resulting behaviour will most likely be detectable by the user as flawed.

As a separate mechanism to alter the execution of actions, IMPULSES are proposed, which can link to different parts of the tree. IMPULSES are similar to pointers and there are two ways of using them within a BT. It either is used to alter and switch whole parts of the executed tree or it can be used as a lightweight logging and debugging mechanism. Combined with BEHAVIORTAGS, IMPULSES offer a way to *hack* the execution order while being able to track those changes. Tags are bit vectors attached to behaviours and allow the developer to track impulses and relevant behaviours. They offer a way to reduce the number of relevancy checks because relevancy is only checked after the tag check is successful. Another interesting presented concept is that the tags can be used to en- or disable large parts of the BT based on a single bit check. This en- and disabling of sub-trees is similar to internet protocol sub-network (IP subnet) masks which hide and structure areas on a network. If the mask is attached to the tree structure, it can present different views of the BT.

The last two concepts contained in BT are STIMULUS BEHAVIOURS and CUSTOM BEHAVIOURS. They define non-leaf nodes which can be inserted into a fixed position in the tree at a later time. The idea behind this is to reduce the number of checks necessary when traversing the tree. The STIMULUS BEHAVIOURS and CUSTOM BEHAVIOURS describe behaviours which are rarely used. Thus, they do not need to be continuously active or even present in the tree. The stimulus behaviours are added for a given

[6]The conference changed in 2015 its title to *nucleai* and scope, integrating now different tracks, including academic ones.

amount of time by an event handler to respond to certain stimulus, e.g. a special event has happened and the player is now able to perform a particular action. The custom behaviours, however, as the name suggests, are added based on some customised call in the logic. Thus, they offer more flexibility to the design. Introducing those two concepts additionally made the execution of the tree more complex and hard to follow for large trees. Isla never explicitly mentions memory as part of the BT. For the Halo AI, he discusses the implementation of a memory system. The memory system, however, is not part of the described BT approach.

As BTs are used by game designers, Isla focuses on providing guidelines to support its usage. He assumes two characteristics of game designers:

- Designers think in terms of triggers—events or situations which drive the agent.
- They prefer to use priorities instead of using numerical values. This, however, contradicts the view of Snavely [2004] who supports the usage of spreadsheets and statistical or fuzzy set design behaviour.

To edit the BT he also reduces the amount of information which should be visible/editable by them. Thus, he creates a designer view of the tree which allows only specific parameters to be changed but not the graph itself. Through the usage of templates and an ontology he additionally reduces the number of parameters which need to be adjusted in the BT. As an illustration, imagine a basic unit which contains most of the behaviours and attributes. A more specialised unit shares most of the parameters. However, it has a set of CUSTOM BEHAVIOURS just accessible to it.

When designing the BT architecture Isla initially focused on four principles: customisation, explicitness, hack-ability, and iterative development. As the graph structure can be visualised and all elements are in place during design time explicitness of the resulting behaviour is given, but this is also true for state machines which offer a similar observable representation. The resulting graph itself is also more organised than a state machine due to the usage of hierarchies and prioritisation. This reduces the number of edges the graph contains in contrast to a state machine. Offering a way to customise and hack the decision-making process is important to game programmers. The BT supports those by using Custom Behaviours and IMPULSES. However, those two features alter the path through the graph drastically, which might impact the understanding for larger graphs. Due to the usage of a hierarchical graph which allows adding of nodes but follows a path from root node to a given leaf node iterative development is easier than using potentially cyclic graphs such as state machines. However, the name BT is slightly confusing as the used structure is not a tree structure as a child node potentially can have two parent nodes. In contrast to a state

machine, behaviours are reused throughout the tree as shown in Figure 2-6 where the Guard behaviour is re-used in several points.

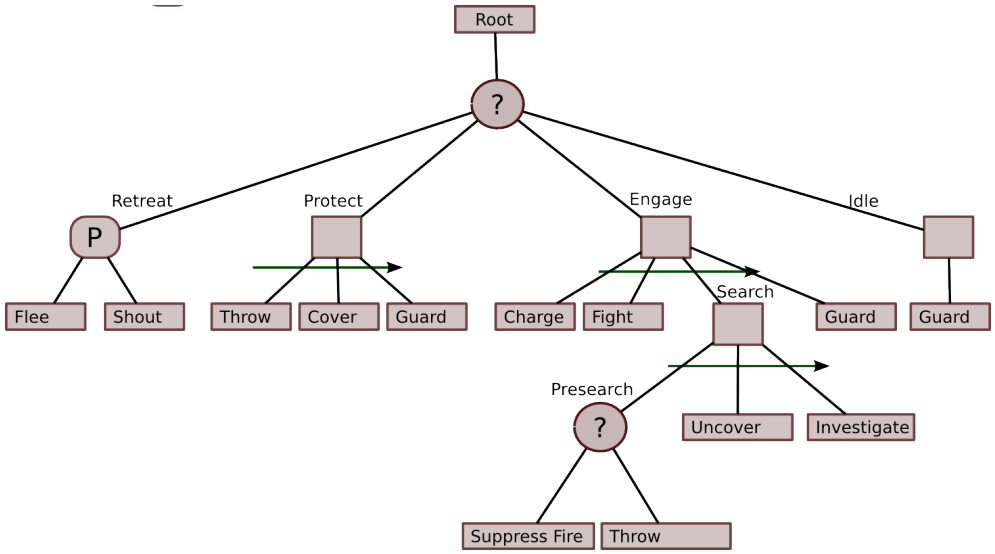


Figure 2-7: A second generation BehaviourTree (BT). The BT features a parallel node for Retreat, selector nodes for Root and Presearch and sequence nodes for the remaining non-leaf nodes. Due to the complexity of Decorators and their impact on understanding the structures, they are left out of this example.

Champandard and Dunstan [2013] extended the BT into what is known as BT version 2 by providing a fixed set of primitives and a more formalised approach. They introduce two types of nodes: leaf nodes which are actions and conditions and non-leaf nodes—deciders—which represent and shape the control structure. They provide great flexibility and a first standardised structure of what BTs are now. A visual representation of a second generation BT can be seen in Figure 2-7. Additionally, all nodes have a set of standard return statements (succeed, fail, running, error) which dependent on the result can alter the execution of the parent node. However, in different implementations [dkollmann, 2011; Kulawardana, 2011; Johansen, 2013; Hamed, 2012] return statements are handled differently, thus, they are not further discussed here. Actions are leaf nodes which execute behaviours such as jumping, firing and so on. Conditions check sensory information from the agents and are used in connection with sequences to block parts of the tree. The deciders are:

- A **sequence** node executes all children in a fixed sequence. Once a child completes its execution, the next child in the sequence executes. If all children are finished, the parent node is finished as well. If a child fails, the sequence fails as well.
- A **parallel** node executes child nodes in parallel, thus they are run at the same

time. It is possible to specify how many child nodes need to succeed or fail for the parent node to succeed or fail. The usage of parallel nodes does not rely on threading as most "current-gen" game systems only feature a small number of cores. Thus, time sharing or scheduling is often used to coordinate between them.

- The **selector** is similar to the prioritised list from version 1. It combines all of the features of the prioritised list into this single node type, making it the most versatile type in BT. Champandard [2007d] however suggests the probability or priority selectors are standard case. The probabilities and priorities are to be adjusted by a designer but not the elements.

- The **decorator** offers a way to augment existing nodes, even at run-time, changing their return value or how they are executed. They provide an additional dimension in customisation to the editor of a tree. However, they increase the complexity of the tree behaviour drastically.

Given those initial node types, game developers started developing visual tools to utilise this mechanism, switching from HFSMs to BTs. The most prominent tools implementing BT are presented in Chapter 2.3. By including a fixed set of primitives the usage of BT has been made easier as they provide a selection of items to choose from instead of following only an approach and each team reinvents their own structure. Additionally, the elements were created with ease of use and behaviour design in mind. They provide a common infrastructure to exchange information and advance the approach as a whole. However, much focus of BT creation is on programmers. They are responsible for creating a valid and robust working tree as discussed by Anguelov [2014]. The trees can be modified and customised by designers. However, they only have access through a limited interface. Anguelov [2014] argues for a shift in attention and a better inclusion of designers into the process of creating and modifying BTs.

Extensions: To include more responsive mechanisms into systems, described by Wooldridge [2009] as multi-agent systems, Bojic et al. [2011] decides on a lightweight approach and re-implemented BT as introduced by Champandard [2007d], instead of using FSMs or more heavyweight approaches. The resulting system was designed for the JADE agent system by implementing a JAVA BT system on top of JADE tasks. However, the result does not include novel contributions to BT or behaviour arbitration itself. It focuses mainly on replacing the currently used model—a state machine—with a mechanism being able to scale better, as well being easier to observe its behaviour. Additionally, as BT is only a technical mechanism to realise behaviour arbitration, Bojic et al. [2011] argues that it is hard to realise state-machine like behaviour with a

BT alone. Their potential future work was aimed at using a hybrid system to integrate state machines and BTs into a single system. This argument shows a missed point in understanding and designing behaviour systems and BT in the first place. BTs were introduced to replace large complex state machines and to tackle the complexity of potentially n^2 state transitions. This shift towards BTs replaced the need to control a large amount of transitions between states. BTs are equivalent in their expressiveness to FSMs. If Bojic et al. [2011] would have employed a more design focused methodology instead of just using a systems design technique, it could have presented them with a way to redesign their agents using BT. This way the methodology could have exposed the functional equivalence of BT and FSM.

A similar academic approach to the second generation of BT, Behaviour-Oriented Design (BOD) [Bryson and Stein, 2001] in combination with Parallel-Rooted Ordered Slip-Stack Hierarchical (POSH) [Bryson, 2001; Gaudl et al., 2013]—a lightweight HTN planner—an approach which will be discussed later on. The approach precedes BT but goes beyond a structural framework as it contains a design approach missing in BT.

Hecker [2009] extends the original definition of BT by Isla [2005] but provides more guidance for system design based on interviews of the original team working with Isla on Halo2. He also introduces restrictions on the tree itself to support the design and understanding of behaviours. Hecker suggests a monolithic tree contained in a single file. This is done to promote the understanding of its whole behaviour. By heavily relying on macros, written in the underlying engine code, he reduces code repetition in the designed tree. To support more modular and separable engine and BT code, he recommends that the tree only uses BT nodes written using the given macros. Behaviours are only linked from leaf-nodes using pointers, therefore a pure design tree is available. In combination with the static, monolithic tree, Hecker can constantly visualise the tree itself while debugging. To support the development he also enriches the nodes and the arbitration process with DebugPrintStates and debug hooks which make it easier for a programmer to grasp the current state of the tree completely at a given time, during execution.

The approaches presented by Hecker [2009]; Champandard [2008]; Isla [2005] focus to a large extent on programmers. They are also extremely oriented on developing behaviours using programming languages. In his blog Anguelov [2014] critically analyses BT from experience and proposes two new advancements, a monitor node allowing the tree to store state information and a gate node offering access to a static tree field accessible by all instances of the same tree. Both advancements are driven by the requirement to instantiate hundreds of BTs for game agents and lead to a BT flavour called Synchronized BT [Anguelov, 2014]. As Anguelov [2014] argues, designers rarely

have a background in programming and by providing only tools for developing agents using those means they are excluded largely from active participation. This introduces a reliance on programmers translating agent designs into the actual behaviour implementation, thus, removing the original designer one step from it. He envisions a visual design language based on BT to support game design. Additionally, Anguelov argues for a transition from using BT only for high-level decision making or the usage of multiple decision-making systems to using BT as the single system for decision-making and the inclusion of a finer granularity in the decision process.

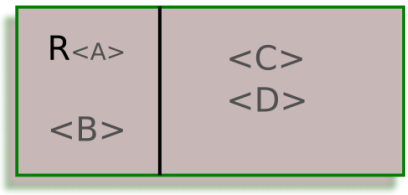
Perez et al. [2011], on the other hand, extended the development approach for creating game agents for the platformerAI toolkit, a two-dimensional game experimentation environment which is discussed later, by using an evolutionary approach and removing the manual designer. Instead of a designer or programmer writing the agent code they employ Grammatical Evolution [O'Neil and Ryan, 2003] to evolve BTs. Their approach focuses on evolving sub-trees which can be re-combined using a constrained grammar. They elaborate in some detail why constraints are required. Their evolutionary search algorithm presents large and hard to read BTs. Additionally, the initial node set they use for the BT contains only high-level actions which are quite abstract and complex. Those actions are created using a lot of design and computation. For example, AvoidRightTrap potentially requires the agent to store information about the world and instantiate actions for multiple runs of the tree. The resulting trees are mostly husks recombining large chunks of logic only optimising their arrangement. Lim et al. [2010] use an approach referenced by Perez et al. [2011]. They apply Behavior-Oriented Design [Bryson and Stein, 2001] to create an initial set of BT agent representations for the game DEFCON. As part of their approach, an evolutionary system is used to optimise those hand-crafted trees and evolve new ones by applying genetic operators such as mutation and cross-over to the initial set. They show that a hybrid approach can be beneficial but finding the right fitness function is difficult as well as evolving agents in games which require a lot of time to run per generation. Similar to Perez et al. [2011] the actions are relatively high level. They are computationally quite complex and the reasoning in the BT is only done on a macro level. Both approaches provide an interesting combination and extension of BT and evolutionary methods. They thereby utilise the evolutionary method to discover novel strategies—BT agents—which might have been overlooked by a designer.

Software Design BehaviorTree (SW-BT)

Before concluding BT, the second approach coining the term was introduced by Dromey [2003]. In contrast to the game BT, this method focuses on software design in general

and not the creation of agents. It also includes a development methodology which guides the user through the process. The approach is very visual and comes with a rich graphical notation in combination with a well-defined grammar. In contrast to the game BT, which is a directed acyclic graph, an SW-BT is a true tree structure. Each node always has one parent or none in the case of the root node. The approach was created to tackle issues in capturing requirements in a single notation. In UML, there would be a need for different diagrams which according to Dromey do not provide the constructive support needed. By capturing the functional requirements for a new product from natural language descriptions in a behaviour tree, it is possible to construct sub-trees for each requirement.

It is important to note that despite the name and the similarity of the visual representation the game and software engineering BT have nothing else in common and should not be treated as related.



(a) A single tree node for requirements behaviour tree. The node can be used in the requirement tree (RBT) and design tree (DBT).

Label & Name	Description
A - Requirement Tag	tag linking to the original requirement
B - Requirement Status	"C" if the requirement is composite, "+" if the behaviour is implied, "-" if behaviour is missing in requirement, "++" if requirement changed subsequently, "@" marks insertion points, explicit behaviour does not receive either "+/-"
C - Component Name	name of the component
D - State/Condition	"[state]" for internal state of the component "??state??" passes to next node when condition for state is met

Figure 2-8: Node description of software design behaviour tree (SW-BT) based on Dromey [2003] which is entirely different from BT. The nodes are combined into sub-trees for each requirement and later on merged into a design behaviour tree containing all functional requirements for a product. A design tree is given in Figure 2-9

The sub-trees for each separate requirement can then be combined in an itera-

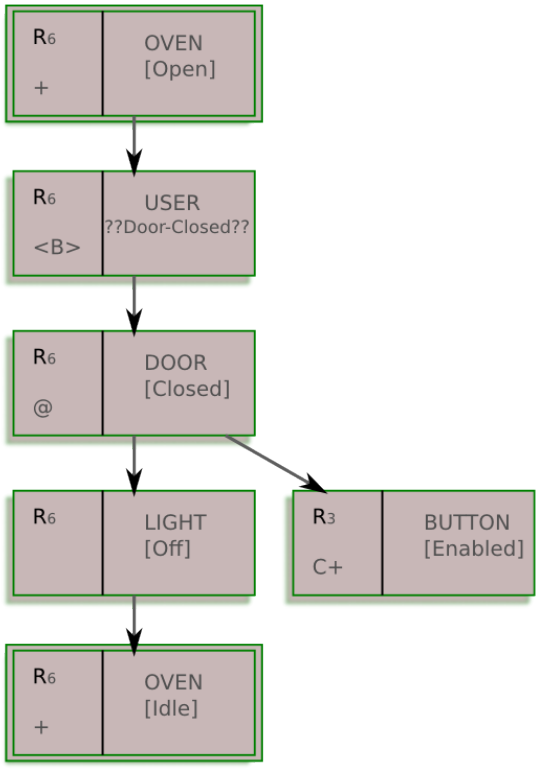


Figure 2-9: A partial tree combining two requirement trees into one partial design tree by inserting requirement tree R_3 into tree R_6.

tive process into a design tree containing all requirements using a "Genetic Software Engineering Method" which Dromey developed. He however completely ignores existing research on other agile software engineering approaches such as SCRUM or XP, which already partition the software into smaller chunks collected on a workspace and recombined dynamically. He claims that:

> Conventional software engineering applies the underlying design strategy of constructing a design that will satisfy its set of functional requirements. In contrast to this, a clear advantage of the behaviour tree notation is that it allows us to construct a design out of its set of functional requirements by integrating the behaviour trees for individual functional requirements (RBTs), one-at-a-time, into an evolving design behaviour tree (DBT). This tree integration very significantly reduces the complexity of the design process and any subsequent change process. [Dromey, 2003, p.2]

In spite of presenting an approach which explicitly contains an underlying strategy for the combination of elements, he states that "conventional engineering" has the

downside of needing to apply a strategy to derive results. Thus, he states "genetic software engineering" (GSE) using behaviour trees is more intuitive and attractive than conventional approaches. This statement is supported by his emphasis of relying only on a single notation for the whole process in contrast to UML or other approaches which combine different notations. When translating the design tree later on into a component interface diagram he also switches notations. Given a missing evaluation against other approaches, this claim can only be taken as a personal opinion. Nonetheless, his approach for deriving a more robust software design and how to identify broken or missing requirements is worth investigating after a rigorous evaluation and comparison to other models. Follow-up research exists [Wen and Dromey, 2004] extending arguments on effectiveness and presenting comparable approaches. Wen and Dromey [2004] elaborate the existing work without using the argument of Dromey that their genetic software engineering and the combination of requirements resembles the gene combination mechanism in DNA.

2.1.2 Spatially Centred Approaches

A large sub-group of games is centred around the concept of a virtual space in which entities have to move and navigate in. Game programmers typically approach the design of those games by picking approaches which they are either familiar with or are stable, fast, well presented and have a low learning curve[7]. The robotics research on spatial navigation is well established and basic techniques are even taught at undergraduate level, integrating them into the tools at hand for future programmers. Additionally, spatial centred approaches are of high importance to the case study presented in Chapter 4 as StarCraft is heavily based on spatial reasoning. Understanding spatial reasoning is one of the fundamental skills of any game programmer and the discussed techniques present some elements which should be part of any game programmers toolbox. Understanding these approaches are crucial in games relying on fast spatial navigation or planning. In chapter 4.3, the design of a Real-Time Strategy (rts) agent is discussed which requires understanding the difficulty of navigating within spatial games. This section tries to provide a useful foundation for this discussion.

Defining a Graph In digital games, it is nearly impossible to search efficiently for paths between two coordinates without using an underlying representation of the world first. This representation is abstracting or mapping original coordinates to a different representational space. In most real-time strategy games the underlying representation

[7]Game programmers generally come from a variety of backgrounds and not necessarily just from computer science.

is tile-based, partitioning the map into either squares or hexagons of the same size. In 3D environments, due to the additional dimension, it is either a navigation point directed graph (NavPoint Set) or a navigation mesh (NavMesh) [Snook, 2000]. NavPoints can be directly mapped to vertices of a graph. For a NavMesh, the approach is slightly different as the representation consists of connected triangles which are connected by their edges, see Figure 2-10. In general, each triangle is considered a vertex in the graph and if two triangles are connected through an edge, the correlating vertexes are also connected. Triangles can also be connected through an explicit edge to allow for more fine-grained control over the navigation.

> Given a graph $G = \{V, E, C\}$ with V being a set of nodes and E being a set of edges. $V = \{v_0, ...v_n\}$ contains a finite known set of nodes which are connected by directed edges $E = \{e_0, ..., e_m | e_a = [v_a, v_b] \wedge a, b \in \{0, ..., n\}, m \leq n^2\}$ between them. The direction of an edge e_a is from the first node v_a in the tuple to the second node v_b. C is the set of cost values for traversing an edge, $C = \{c_0, ..., c_m\}$ where c_a is the cost to traverse e_a. An undirected graph can be treated as a graph containing edges where for each edge $e_a = [v_a, v_b]$, with an allocated cost of c_a, there is an edge $\dot{e}_a = [v_b, v_a]$, with an allocated cost $\dot{c}_a$, and $c_a = \dot{c}_a$.

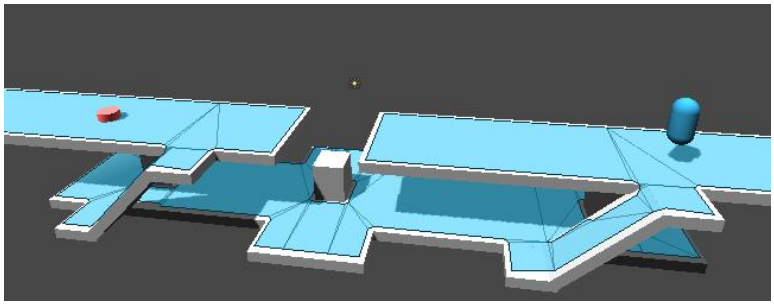


Figure 2-10: A NavMesh, represented in blue, allows an agent to move in a given environment by using the underlying triangles for navigation instead of a node or grid-based system. Unity automatically generates the NavMesh based on a given static geometry for the environment. NavMeshes currently are the standard approach to navigation infrastructure in games.

Shortest Path

Dijkstra's shortest path algorithm [Dijkstra, 1959] and A* [Hart et al., 1968] belong to the foundation of most computer science degrees. Most of the implementation used today are extended versions of the originally designed algorithms. A* is probably the

single most commonly used technique in games when searching for a path between two points in a graph.

Both shortest path implementations present challenges which make them harder to use in practice than initially anticipated. Dijkstra's algorithm is considered greedy when it comes to the nodes which are checked. It checks all the nodes which are on the way from source to target. For large environments, this can get quite expensive. Guided by a heuristic function, A* only searches a smaller subset of nodes on its way; this makes the approach more appealing as it still finds the optimal route. Fundamentally, both approaches work well in static environments and need to know the whole environment to be available for checking the nodes. They both require recalculating the whole graph when searching for a path or when the environment changes. As games are highly dynamic, the environment often changes, so a path needs to be recalculated every time which can become quite costly.

As A* is applicable to graph problems, it can also be used for other pathfinding problems such as planning a trajectory through space. Orkin [2004] introduces A* in his planning system to efficiently find a path from a given goal to the needed action, while exploiting the existing research on optimised heuristics for A*. As the goals change or the pre-conditions for given actions are inaccessible, re-planning is required which introduces similar problems as the dynamic environments into the search. Assuming that the search space for planning is sufficiently large, this recalculation becomes quite costly.

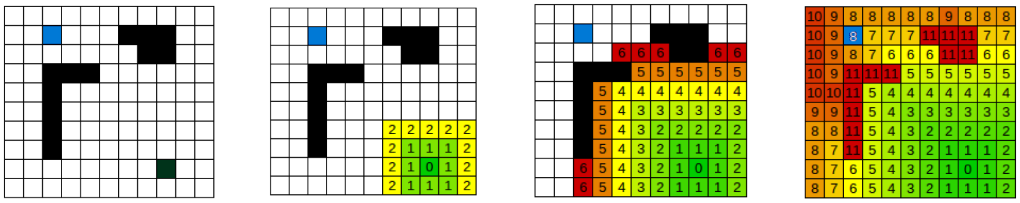
Koenig et al. [2004] present an analysis and reflection upon that argument drawing in knowledge about the state of the art techniques and the requirement of robotic environments and games to recalculate a path in a non-static or changing environment. They illustrate the weakness of A* based approaches such as their own Lifelong Planning A* (LPA*). A* is heavily based on heuristics to guide the exploration around the path from source to target. In contrast to Dijkstra's shortest path, this approach visits fewer nodes on a graph making it a fast technique. If the cost c of an edge changes the whole graph needs to be recalculated as A* has no way of going back to that node. To not re-plan the entire graph LPA* uses inadmissible heuristics and memorises previous plan and the plan creation process. To guarantee a fast solution the search area around the previously calculated path is now explored. This, however, impacts the resulting path. Koenig et al. [2004] argue that a way around that is the usage of a truly incremental search approach such as Dynamic A* search (D*).

Stentz [1994] introduce D* as a successor of A* and as an iterative search algorithm able to deal with a changing C for a graph $G = \{V, E, C\}$. It works on maps with partial or no initial information and derives the optimal solution. Functionally,

it uses the initial information to plan an optimal route and then uses changed information (discrepancies from the original cost) to recalculate a new optimal path. This recalculation is done in a more efficient way than re-planning with A* or using LPA* and is one of the dominant search approaches in changing environments. Having said that, in games A* is still heavily used due to insufficient knowledge transfer about the benefits of more dynamic ways to calculate paths.

Potential Fields

Potential Fields (PFs) are one type of gradient technique which originates from robotics research [Krogh and Thorpe, 1986; Khatib, 1986; Barraquand and Latombe, 1991; Konolige, 2000]. It is inspired by physical forces such as gravity or magnetism. Simply put, the underlying mechanism of the approach is that an entity navigates through an environment by utilising attracting and repulsing forces. It moves from a high potential towards the lowest potential, whereas the goal should have the lowest potential in the environment. Potential fields are appealing to robotics and game development due to mathematical elegance and simplicity [Ge and Cui, 2000].



(a) Initialisation: A navigation map containing start location (blue), target location (green) and obstacles (black).

(b) Iteration 2: After the second expansion of the goal node.

(c) Iteration 6: After the sixth expansion of the goal node.

(d) End of Expansion: All cells have been assigned a cost. All obstacle cells get a cost larger than the maximum of all other cells.

Figure 2-11: Figures (a) to (d) show the expansion of a potential field in a 2D map. During this process, all map cells will be assigned a cost which in turn is used to create path vectors from cell to cell. These vectors allow an entity starting at a random location on the map to move down the gradient towards the minimum.

According to Krogh and Thorpe [1986], regular planning approaches to robotic navigation, such as A* or similar shortest path algorithms, do not incorporate the vehicle dynamics into the problem. They force the robot to drive through each node from start to goal along a path which might incorporate sharp angled corners. Those sharp angles lead to jagged trajectories where the robot has to slow down and speed up, resulting in non-optimal behaviour. Additionally, non-incremental approaches to planning require re-planning of the whole path whenever the environment changes.

They could even be not applicable when traversing unknown terrain. This can lead to expensive CPU costs resulting in energy inefficient robot control or non-optimal solutions. A more detailed discussion on that is given by Konolige [2000].

Naive Potential Field Approach: To be able to utilise a potential field, the given environment needs to be discretised by generating a representative data structure such as a graph or matrix from it. The coarseness of the representation is a factor to consider during this step as it is affecting the solution in terms of accuracy. The coarseness also affects the computation time and the amount of memory used. Typically, the resolution of the map does not need to be finer than the minimal dimension of the smallest movable unit in the environment, e.g. the robot or game character for which the potential fields are used. Thus, a potential, minimal gap between two obstacles is still sufficiently visible for fitting the entity through. The discretisation can be either done by introducing a grid structure on top of the terrain or creating a distributed set of nodes or by applying similar techniques to abstract a non-discrete environment. After obtaining a data representation of the environment, the next step involves an update algorithm for its contents.

Expanding potential force algorithm (wavefront approach) [Khatib, 1986]:

1. The goal cell receives an initial cost of $c = 0$.
2. We set our counter $n = 0$.
3. We add the goal cell to a new empty list we call wavefront.
4. We select all cells from the wavefront with $c = n$.
5. We assign their nearest neighbours who are not yet in the wavefront and are not obstacle cells with a cost value of $c = n + 1$ and add them to the wavefront.
6. we increase n: $n++$
7. If we have not visited all cells in the grid we go back to step 4. If all non-obstacle cells have been visited, we continue.
8. For each obstacle cell in the grid we assign a cost $c_o = \infty$, or $c_o = max(c_i) + 1$ for all non-obstacle cells i in the grid.

In a uniform rectangular grid, this approach computes the minimal Manhattan distance from a cell to the goal cell [Konolige, 2000]. The resulting grid now represents a distributed cost with a gradient towards the goal. In a fixed environment we now can position entities and if they follow the gradient from cell to cell, from highest cost

to lowest cost reaching a minimum, they will ideally reach the goal. However, this approach has some known limitation addressed in works by Krogh and Thorpe [1986]; Ge and Cui [2000]; Hagelbäck and Johansson [2008]; Konolige [2000]. As you expand the cells, it is possible to create local Minima within the grid. Those Minima occur in concave shaped obstacles and can lead to dead ends for the entity. Additionally, if the resolution of the grid is high, the computation of it is extremely high as you need to visit all cells within the grid. With the current description, the impact of the repulsing forces is quite low as it currently only repulses the entity one step before the object but in actual settings obstacles might have a repulsing force affecting neighbouring cells as well to guarantee that the used robot does not collide. This can be done by including either a Gaussian or linear decreasing gradient expanding beyond the obstacle. Another major issue of the presented approach is that entities are prone to graze the corners of objects which can be addressed using the previously mentioned extension of the repulsive force.

Hybrid Potential Fields: Krogh and Thorpe [1986] present an approach to robotic navigation. They combine potential fields and high-level planning. Thus, they avoid the negative impact of local minima on finding the correct solution. The approach is split into two phases, a high-level planning from start to goal and a second phase which takes into account only local information to navigate around obstacles and respond to real-time sensory information. As discussed before, path planning is expensive and reduces the flexibility of a robot or in the context of this thesis the time each agent has for reasoning. Thus, Krogh and Thorpe [1986] argue for sparse usage of planning. He proposes the usage of a coarse map to do rough initial planning using A* and the generation of a set of critical points which define the initial path towards the goal. The number of critical points in the set is not specified, but should be kept as minimal as possible to reduce computation costs. The next step is to relax the position of all critical points to minimise the cost of travel along them. The cost for a point does not need to reflect only the distance of travel but can also take the terrain into account or the closeness to other objects (repulsive force).

The update and relaxation of critical points are a recursive process until no points have been relaxed on the last cycle. Within one cycle a node can only be relaxed once and only by one unit. A node can only be moved perpendicular to a line through its two enveloping nodes.

This can normally be done before the entity has started moving and only needs updating if drastic map changes happen. Thus, the robot is able to amortise the cost of those expensive computations over time.

The second phase is the dynamic steering which takes the real-time sensory information into account to create the trajectories towards the goal. The advantage over other potential field approaches is the potential fields are position and velocity dependent, whereas other approaches are only velocity dependent. Meaning, only cells are computed which are on the path towards the goal instead of the whole terrain. Thus, issues relating to local minima and high computational costs for large maps are addressed as you only look along the relaxed path and intermittent steps. To calculate a potential $P(x, v)$ for a critical point you use its current position $x = [x_1, x_2]$ and the velocity at that location $v = [v_1, v_2]$ where

$$P(x, v) = P_G(x, v) + P_O(x, v) \tag{2.1}$$

is the potential function for any given critical point in the environment. This is done by taking the attractive potential towards the next element P_G in the critical points set and the repulsive potential of obstacles P_O on its way into account. $P_G(x, v)$ is defined by the minimum time to reach the critical point x from the current position. Further details on how to efficiently compute P_G in a dynamic environment are given by Feng and Krogh [1986] as it goes beyond the scope of this chapter[8]. The repulsive potential of obstacles is defined by

$$P_O(x, v) = \frac{v_O}{v_O^2 - 2\alpha x_O} \tag{2.2}$$

where x_O,v_O and α are: the directional vector towards the visible obstacle, the velocity towards the obstacle and the maximum steering angle of the entity. To reduce computational cost, P_O is only computed for the obstacle along the current velocity v vector. The next critical point is chosen once the entity reduces its velocity towards the current point. If a point is not visible from the currently active critical point, an intermediate point is inserted into the set. The intermediate is chosen as the corner of the obstacle blocking the visibility closest to the originally selected point. The result of this real-time two-phased approach is a computationally tractable approach taking the entities capabilities to manoeuvre in the given environment into account and minimising the total cost to arrive at the global goal. Konolige [2000] proposes a nearly identical approach to Krogh and Thorpe [1986] supporting the applicability to real-time spatial movement with robots.

In games, the limitations for entities are not as rigid as in real physical environments. Thus, the presented hybrid approach would not be required for optimising smooth trajectories. However, creating the impression of physical limitations and rigid

[8]In games a linear approximation between the current position and the target position given the current velocity should, however, be sufficient.

movement with a computationally tractable approach should be appealing as it aids the desired impression of a given virtual entity.

Bourg and Seemann [2004] and Olsen [2002] introduce potential fields into games as an attractive computational approach for game unit control that can be applied on a global scale. The approach does not go beyond what was previously introduced as naive potential fields. In the case of game AI, these PFs are described as being useful to attract units to weak and desirable targets while repelling them from stronger more dangerous forces. Additionally, a single function can be used to provide a large amount of control, which makes them appealing. By modifying the variables of the repulsive and attractive gradients for each force, a way can be provided to include learning and improvement into an AI mechanism. Hagelbäck [2012]; Huang [2011] describe approaches of using potential fields in digital games with great success. Hagelbäck and Johansson [2008] introduce their approach to multi-agent control using potential fields (MAPF) in real-time strategy games (RTS) and achieve results which are able to compete with A* approaches in path planning. A result which brought them industrial attention featuring their approach in the AiGameDev tutorial set[9]. By utilising a genetic algorithm (GA) to optimise the parameters for different unit types and their related potential fields Sandberg and Togelius [2011] extend the capabilities of MAPF offering a way to include offline learning into the game development process.

Potential Fields, in contrast to Shortest Path approaches, offer a low-level control mechanism for robots and game agents. They do so by giving a visual representation of the reasoning process to the user. This low-level access and the lower computational cost makes them appealing to game developers and designers.

Influence Maps

Schwab [2004] introduces Influence Maps (IM) in his collection of game programming techniques not as a single technique which is directly applicable to game AI but as a helper technique [Schwab, 2004, pp.373]. Due to its very visual structure and flexible, straightforward implementation, it is getting more and more popular in game development [Tozour, 2001; Sweetser, 2004b; Schwab, 2004; Millington and Funge, 2009b]. Tozour [2001] emphasises in his introduction to influence maps for game programmers, that IM is not a single technique but a more general approach of how to use spatial data in games to aid the decision making.

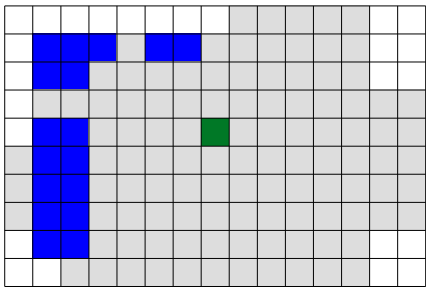
On a basic level, an IM consists of a simple array/matrix structure containing cells

[9] AiGameDev is organising industrial workshops and conferences and manages an online collection of techniques which are often used as a first access point for game developers when searching for new or useful techniques.

related to information about the game world. Those cells are correlated with spatial locations from the game world. The primary function of an IM as used in games is similar to a cluster of place cells from Neuro-Science [Cools, 2012], their purpose being the storage of information about the world bound to specific locations. In this respect, both models are quite similar as they bring an abstraction layer between real world information and decision making by applying content and context dependent filters to the world. A cell value in the map is, in the simplest form, a numerical value representation of its influence[10]. In some cases, it can also be a collection of more complex attributes which are used to aid the decision-making process. The values of a cell can represent different types of information: thread level, terrain height levels, visibility of units, or other terrain factors affecting attack or defence values of units. Those values can be used to create different layers of the IM, which allow a more fine-grained analysis. An important feature of the IM is to give access to the relation of influence between different cells without having to analyse the world each time, a process that is crucial in decision-making.

Depending on the content of a cell or the complexity of the map, the influence can also bleed or spread into connected cells. In this, the process is similar to potential fields. An example would be a wall which cannot be passed. The wall would not spread its influence to connected cells when it comes to walkable terrain but on a different layer, like visibility, the wall would spread its influence casting a shadow over connected cells. Influence maps are often represented as heat maps for visual analysis, which makes their information easy to understand as you can, for example, represent thread levels in a colour gradient from green to red. The visual representation can, same as the IM underneath, be layered to allow for different purposes to be visualised independently which can aid the design and understanding.

[10]A better term for influence in most cases would be importance to the agent.



(a) A simple visual representation of an influence map. In grey, the spread of creep, in blue, the available resources and in green, buildings to protect are shown. The map is based on figure (b), abstracting the most important information about the world into it. The actual IM could be a two-dimensional int array containing int values for the important elements encoding their importance to a bot.

(b) StarCraft Zerg central building oozing creep using a Gaussian-like distribution. The creep is a visual game mechanic which can be represented by an influence map.

Figure 2-12: Influence map

Due to its simple structure, the approach can be scaled to different requirements even 3D and real-time. In the case of 3D environments, positions in the world could be replaced with navigational mesh cells (NavMesh cells Snook [2000]; Tozour [2001]). To visualise more complex maps, the mentioned layers can be combined. This is often done using a weighted sum based on a designer decision of the thought of importance of specific features. With the weighted sum for each cell, complex attributes are compiled into a single desired measure that can be visualised in the map as discussed before. The design of the weighted sum function is, however, non-trivial in most cases. The process is in some cases more complex than optimising certain parameters; sometimes it also has to involve user testing. The combined map offers a way to visualise complex relations about world information. That information can aid finding unique features in the world such as navigational bottle necks—choke points—or safe zones. The information about the world in form of an IM are generally used for:

- **Terrain analysis** in real-time strategy games uses influence maps to determine which positions are of strategic value or potential danger.
- **Path finding** can be aided by providing a "good" cost function or heuristic for the path planning approaches to calculate the best path for an agent. Here different layers of the IM can represent different criteria for what "good" represents for a designer or a specific agent.

- **Ground control** is sometimes used as part of the game mechanic for certain games to determine which player controls a map region. An example is the creep spread in StarCraft which can be represented by an IM, see Figure 2-12b.

Figure 2-12a shows a local IM for a Starcraft agent. The map has a relatively low resolution compared to the resolution of the game. Lower resolution maps can be beneficial to reduce space and computation time or when only focusing on some important points of the game environment or when using it only for a given location. The resolution of a given map may depend on different things like the world size, the needed accuracy of the influence of a location, or simply the memory limits for the AI approach. They are an abstract model of the environment incorporating the given circumstances.

An important aspect of any technique used in games is its impact on the performance and the amount of computational power and memory required. If the resolution of the IM or the number of layers is high and the world map is large, the memory and CPU cost of maintaining and updating the map is quite high. For standard approaches, the cost is $O(nm)$, where m is the number of cells in the map and n is the number layers or attributes. Let us use StarCraft as an illustration. The maximum map size is $m = 256 \times 256 = 65536$. As StarCraft is played in real-time, this means for a naive implementation, the map is updated potentially 30 times per second to incorporate changes. In those cases, it makes a large difference if your IM has 1 or 10 layers/attributes per cell as each additional layer doubles the amount of checks or updates. Millington and Funge [2009b] discuss this issue as well and present different influence models to counter the cost of large updates. An important connection they make is that a single layer of the influence map can also be interpreted as a pixel image. Thus, image-based approaches to blur and sharpen an image can be used as well. The blurring of a pixel is similar to spreading the influence to connected cells but works over the whole image and as it is a matrix operation can be done on dedicated hardware. This removes some pressure on the CPU.

Influence Maps in contrast to the previously introduced potential fields are not used to control or navigate units directly. They are more beneficial to providing a more versatile tool either for designing levels or in understanding weaknesses in their design. They can also be used to aid the decision-making system or path planning. Nonetheless, it is important to keep the cost of updating and checking the IM in mind to not increase the CPU strain. They are often used in games to calculate defence points, visibility of or by enemies, or where to best set up for ambush points.

Sweetser [2004b] integrates IMs with Neural Networks to replace the weighted sum—the selection mechanism to weight the attributes and layers. Her approach is

not focusing on design feedback but on providing the reasoning mechanism of an agent with an additional abstraction layer instead of using the raw world knowledge. To control the agent, she uses Neural Networks discussed in Section 2.1.3. On one hand, the general idea of optimising the calculation of how to combine the different layers is interesting and can optimise the map cells. On the other hand, the computation workload and the testing are quite intense. Additionally, the designer has less fine control over special behaviours directed by this black-boxed map. Nonetheless, the computation is only intense during the learning phase of the Neural Network and NNs are a standard approach which works well out of the box. Once the game developers are satisfied with the resulting combined IM, the Neural Network only needs to be re-trained if the underlying data set changes. This, however, can happen quite often during development. New features are constantly being added, leading to the need for an extra function in the early phases of development. Sweetser [2004b] also argues that this can be a positive feature as it allows the AI to adapt to specific players after shipping the game. This statement, however, is questionable, taking the discussion of Thompson [2014] and the general arguments made at industrial events into account. Games need to be as stable as possible and game developers would rather not use a feature that can lead to a potentially negative user experience.

An interesting, yet only focused on designer support, approach is presented by Tremblay et al. [2013]. In contrast to the previously shown work, Tremblay et al. [2013] provide a tool to understand and model game levels based on player path analysis. The tool allows a designer to include a game level and agents within the environment plus their intended movement patterns. The developed tool then calculates all possible paths a player can take. For the calculation of possible paths they use Rapidly Exploring Random Tree (RRT) [Lavalle, 1998] to dynamically integrate the information from the changed environment such as the moving NPCs into a set of initial expanding graphs. Instead of simply showing a multitude of paths, they break the paths into smaller segments and clusters close segments according to their distance. According to the authors, this produces a heatmap-like representation of the game level. What it does is deriving an influence map including traces where players are potentially able to go. The designer is now able to interpret the visual feedback about potential player paths and can alter NPC positions or the general layout of the level. This allows adjustments to possible paths. It also allows the designer to see if designed areas in a level are unreachable or would rarely be visited.

2.1.3 Evolutionary & Learning Approaches

For most naive developers, the expectation for evolutionary or learning approaches is that they are useful for replacing the design of games or parts of the game design with automated search approaches. However, currently the biggest contribution those approaches can provide is robust optimisation and augmentation of subsystems in games, as illustrated by Sandberg and Togelius [2011]. Computational Creativity, the field of AI researching the fully automated generation of games or game elements, is relatively young and focuses on exploring the merits of understanding and automating the creative process [Boden, 1998; Colton et al., 2012; Cook et al., 2014; Wiggins, 2006].

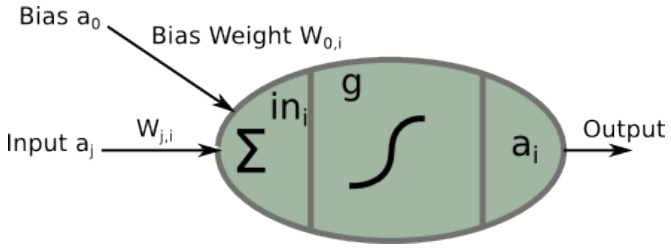
However, this state of the art research has not managed to arrive in the industrial sector, with a few exceptions as discussed below. In chapter 7 a new perspective of using evolutionary techniques is discussed which relies on the foundation of this section and extends it.

Artificial Neural Networks

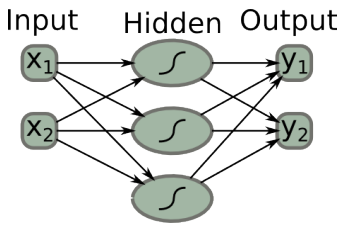
The *artificial neural network* (ANN) approach is based on the attempts to replicate the neural systems of the brain [Russell and Norvig, 1995, pp.764]. The approach is based on a model of a single neurone, see figure 2-13, and favours parallel computation due to distributed computational task of neurone in the network. Neural networks incarnate of learning and adaptive systems based on the previous arguments. There are two main types of ANNs, cyclic (recurrent) neural networks (RNN) and acyclic (feed-forward) neural networks. Feed-forward networks are the go-to solution when it comes to employing neural networks. They are easy to understand and implement and allow the modelling of complex functions in a data-driven way. The advantage and downside of feed-forward networks are their static nature once their training is done. The standard approach for training feed-forward neural networks is using back-propagation using a test set of known inputs and outputs until the measured error over the test data is sufficiently small.

Feed-forward networks and the training using back-propagation allow the developer to follow the signal and error flow as the network does not contain states and produces the same output for a given input.

The ability to understand the signal flow decreases drastically with either the size of the network or when RNNs are used [Russell and Norvig, 1995]. As the size of the network increases the number of connections between each of the layers increases by the power of two for a feed-forward network. Recurrent networks are more flexible and more closely model biological neurones and how they link in the brain to a greater



(a) A model for a single neuron based on Russell and Norvig [1995]. The neuron has an input signal a_j for each connected neuron j from an earlier layer and an a connected weight W_j, i to modify the strength of that signal. The neuron also has a weighted bias assigned to it which can be used to modify the activation function g of the neuron.



(b) A feed-forward neural network containing an input layer with two neurons a hidden layer with three and an output layer with two neurons.

Figure 2-13: Figure (a) shows a basic artificial neuron defined by input signals, their weights, the input function in_i, the activation function $g(in_i)$ and the output value of the neuron ai. Figure (b) utilises the artificial neuron on three layers defining a simple artificial neuronal network.

degree [Russell and Norvig, 1995]. RNNs in contrast to feed-forward networks allow connections directed to earlier layers. Hopfield networks [Hopfield, 1982] for example, contain bidirectional connections using the same weight in both directions. RNNs due to their cyclic structure offer a way of including state or memory into their decision process. This makes them not only more powerful but also harder to control [Russell and Norvig, 1995; Millington and Funge, 2009a; Bengio, 2009].

Typically, artificial neural networks are quite shallow, containing only a small number of hidden layers due to the amount of weight adjustments needed before converging. With the breakthrough by Hinton in 2006 deeper networks became a possibility. In contrast to previous approaches training the whole network, Hinton applied a greedy approach training one layer at a time. This was done using unsupervised learning for each layer using a Restricted Boltzmann Machine. A deeper analysis into Deep-NN is given by Bengio [2009].

Industrial publications on Neural Networks [LaMothe, 2000; Millington and Funge, 2009a] usually try to include a full overview of them by presenting the whole field in a primer. This primer is done with a focus on implementation coupled with a general concept description. However, this approach over-simplifies in cases, for example, LaMothe [2000] presents the recurrent networks developed by Hopfield in the same context as feed-forward networks including only a walk-through approach for the underlying algorithm. LaMothe [2000] states that a Hopfield net is an iterative auto-associative memory but does not clarify the meaning. What Hopfield introduced was a way to retrieve the correct memory state given a partial or approximated version of it.

Nonetheless, from an application point, he emphasises on the difficulty of employing such an approach as it might not return the desired information at a fixed time, a general concern of RNNs. Instead of providing a full description of all types of ANNs Sweetser [2004a] gives a lightweight introduction to the concept of artificial neurones which is nearly identical to the one given by LaMothe [2000]. She then focuses on providing C-style implementation details for feed-forward neural networks and design decisions for modelling them. However, those decisions are a step by step list, only presenting options without sufficient reasons for choosing them.

> There are many choices to make when designing your NN. [...] Second, you need to decide what types of units will be used, such as linear or sigmoid, by choosing the activation function that you will use. The example in this article used a type of sigmoid function, namely a logistic function. This function is commonly used in networks that use backpropagation learning. [Sweetser, 2004a, p.622]

The given example for the discussed step misses any support for deciding upon which alternative type of neurone the network should have. Her argument that back-propagation learning is often used in conjunction with the sigmoid function is true but does not aid the decision process rendering the steps less helpful. Leaving introductory contributions to neural networks aside. Comparing her discussion with Millington and Funge [2009a]; LaMothe [2000] presents a disparity in academic and industrial focus. Her argumentation does not take actual implementation and restrictions of games into account which are a constant reminder of industrial focused work.

In current games, the usage of ANNs is similar to the introductory models described above and the most prominent case of a still learning neural network use inside a deployed commercial game is the *creature* in Lionhead's Black&White [Millington and Funge, 2009a]. The network is trained by player feedback when rewarding of punishing the creature for actions performed in the game world. Another example is the AI controlling racing cars in Codemaster's Colin McRae Rally 2.0, which models the parameters of the track and specific cars [Togelius and Lucas, 2005].

Evolutionary Algorithms

Evolutionary Algorithms (EA) extend the metaphor of biological evolution into the domain of computation systems [Bäck, 1996; Schwefel, 1993]. They thereby borrow concepts from genetics such as genes, mutation, or recombination to describe a process of adjusting programs or program parameters in an incremental process. The field of

EA can be subdivided into four areas to emphasise different aspects of the evolutionary process. The initial sub-division is based on Bäck et al. [2000].

- Genetic Algorithms (GA) generally use bit vector representations for the individuals. Thus, functions are enabled or disabled depending on their state in the vector. Recombination is used as a main operator for creating new individuals and mutation is only used with a small percentage as a "background" operator but not as a driving force for the evolution. The selection scheme is probabilistic.

- Genetic Programming (GP) as introduced by Koza [1992]; Poli et al. [2008] is a form of EC which on the first glance is similar to GA. Recombination is used as a main operator and mutation is only a secondary operator when evolving new individuals. The selection process is probabilistic and the elements in the individuals vector can represent functional elements. However, the vector can also contain parameters and it is normally not of fixed length containing pointers which can change location on the vector. Thus, GP evolves functional computer programs instead of parameters or functional representations. This exploration, in practice, opens a larger search space than any other EA approach.

- Evolutionary strategies (ES) generally use real-value vectors to address attributes or parameters rather than switching only individual functions on and off. They use normally distributed mutation and recombination as equal operators for creating new individuals and use deterministic offspring selection, for example using a rank-based selection.

- Evolutionary programming (EP) works, same as ES, on real-value vectors. It does not utilise recombination and solely applies mutation as a way of altering individuals. The selection operator is probabilistic and the approach was originally developed to evolve FSMs as mentioned by [Bäck et al., 2000, chap. 18].

Underlying all Evolutionary algorithms are similar principles which unite them and a generic framework can be given in the following form [Bäck et al., 2000, chap. 7]. Using I as the search space containing all individuals $a \in I$ and $F : I \to \mathbb{R}$ the fitness function assigning real-valued a fitness value to each a. The size of a population is specified by μ for parent population and λ for offspring population size. $P(t)$ represents a given population at time t and consists of individuals of type a. To alter a population mutation, recombination and selection operators may be utilised each with specific characteristics Θ.

For all four mentioned EA approaches the actual algorithmic steps within the loop of evolving new individuals are similar and based upon the biological concepts evolution. Before starting the evolutionary process, the time $t = 0$ is set to track the evolution of the population $P(t)$. The population $P(t = 0)$ is initialised with either random or predefined individuals $P(t = 0) = \{a_0(t = 0), \ldots, a_\mu(t = 0)\}$. The initialisation is a crucial step for each EA and differs on the actual approach. Depending on the modification criteria Θ_r, Θ_m an initial evaluation of the pool is carried out to assign each individual a a fitness value.

Evolutionary Process:

1. (Recombination)
   If this step is applicable a new population is created $P'(t) = recombine(P(t), \Theta_r)$ using the parent generation and the specific criteria for creating a new population $P'(t)$. Such a criteria could be the percentage of parents which are used for recombination κ, or if the recombination is using single or multi-point crossover.

2. (Mutation)
   If a new population $P'(t)$ was created in the previous step then a third population $P''(t)$ is created by taking each individual $a'(t) \in P'(t)$ and exposing it to the possibility of mutation $a''(t) = mutate(a'(t), \Theta_m)$ putting the exposed individual

into $P''(t) = \{a''_0, \ldots, a''_\lambda(t)\}$. If no recombination took place the approach uses $P(t)$ instead of $P'(t)$ to create $P''(t)$.

3. (Evaluation)
   During this step the population is evaluated by calculating the fitness of each individual taking the total number of offspring into account. $F(t) = evaluate(P''(t), \lambda)$

4. (Selection)
   During selection set of offspring is chosen to go into the new generation as λ can be larger than μ. The new population $P(t+1) = selection(P''(t), \Theta_s)$ is created from $P''(t)$ and additional criteria are applied Θ_s according to the used approach. A criterion could the applying co-variant parsimony pressure Poli and McPhee [2008] which is used in GP to counter bloat in offspring generations.

5. (Clean Up and Increment) All elements which have not made it into the previous generation are cleared and approach dependent measures are taken either to save the fittest individual or insert specific individuals into the new generation by force. Additionally the generation counter is incremented.

This presented algorithmic approach is basic but is applicable to all EAs and similar steps are often referred to when creating a new EA system as Poli et al. [2008] suggest in their field guide. In the industrial literature [Schwab, 2004, chap. 20] discusses EA in some depth. However, he refers to it as Genetic Algorithms mixing GA,ES and EP. He presents a basic approach similar to the EA one given above, but supplies C-Style code examples and more importantly a list of pros and cons for using EA. Statements like "GAs are stochastic methods and are considered a form of brute-force search."[Schwab, 2004, p.452] generally do not aid a decision process for choosing GA/EA and create a negative co-notation for the approach, whereas a thorough exploration of the feature space is in some cases beneficial. The included points he makes are high level and are similarly fitting for most learning approaches. Thus, they are too general to be useful for specific versions of EAs and their applicability.

Lucas [2005] presents an approach using an evolutionary algorithm instead of back-propagation to modify the weights of the network. The presented approach is motivated by trying to gain insights into what effective approaches for digital games are, in contrast to existing research on chess and checkers. Thus, exploring reactive neural networks presented an interesting starting point. An issue emerges when interpreting the behaviour of evolved networks. To understand the behaviour of the agent controlled by a ANN the individual weights need to be analysed or the exhibited behaviour within the game needs to be observed. The first option is in most cases only possible for small

networks due to the sheer number of weights. The second option is time consuming and highly interpretive.

Stanley et al. [2005] present a similar approach to Lucas by evolving neural nets. The main difference is that they use their approach as a core game mechanic in the game NERO. In the game, the player is in charge of selecting and evolving artificial soldiers to fight in teams against other teams. The approach underlying this game is Neuroevolution of Augmenting Topologies (NEAT) [Stanley and Miikkulainen, 2002]. In contrast to the previously described neural networks, the connections of the hidden layers, the amount of units in the hidden layer and the structure of the hidden layers are altered fully automatic. This allows the evolution highly structured and optimised networks. An interesting concept used in NERO is the human controlled fitness function. This player controlled function allows online modification and direction of the evolutionary process.

Other games such as *Creatures* [Grand et al., 1997] evolve simple neural networks for control and learning in the virtual world. For example in CREATURES a NORN—one of the creatures inhabiting the world—has a genome associated with it. This genome allows for recombination and mutation when breeding new creatures based on pairing desired strains. The ability to create new individuals and to watch them learn skills from each other are the main aspects of the game. A more research based proof of concept and comparison work of employing ANNs in game controllers for racing games is given by Togelius and Lucas [2005]. The resulting automated drivers are far from being able to compete with human players but show weaknesses and potential in ANNs and machine learning in general. However, keeping their results in mind, all forms of machine learning require careful design and sound understanding on the developer's side in order to reduce the combinatorial possibilities of the behaviour to be learned. This is supported by Lucas [2005] who concludes in his work on MS. PAC-MAN that ANNs might not be the right computational approach for this particular game, a statement which is rarely mentioned in literature.

Monte Carlo Tree Search (MCTS)

MONTE-CARLO TREESEARCH (MCTS) is a relatively new algorithm introduced by Coulom [2006] in 2006. It combines the precision of tree search and the explorative power of Monte Carlo sampling. Thus, being able to balance exploitation of the current search with exploration of other search areas. Browne et al. [2012] provide an exhaustive overview of the state of the art on MCTS covering algorithmic description and application areas including weaknesses and strength of the approach.

"MCTS rests on two fundamental concepts: that the true value of an action may be approximated using random simulation; and that these values may be used efficiently the policy towards a best-first strategy." [Browne et al., 2012, p.5]

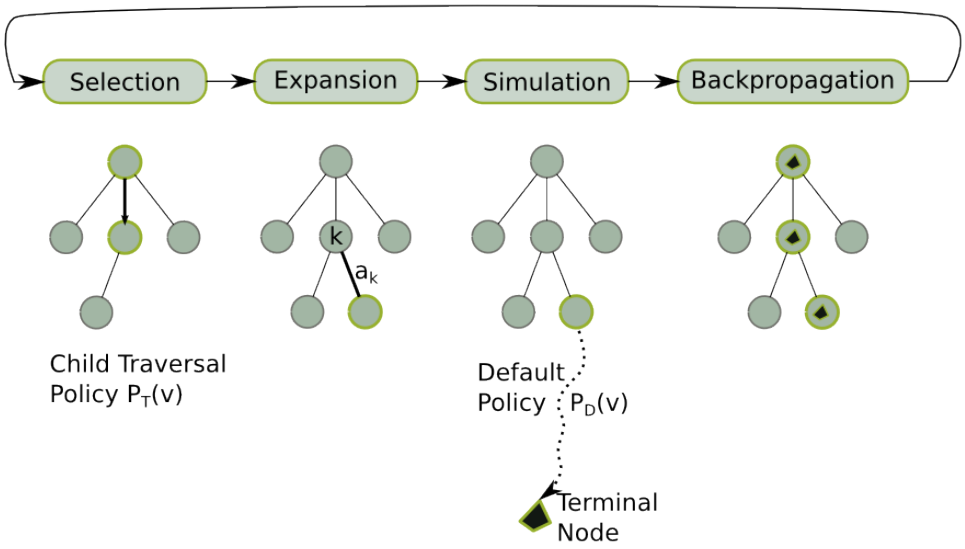


Figure 2-14: A search tree expansion illustrating Monte Carlo Tree Search steps on a high level. The four main steps are visualised on a graph example which expands asymmetrically along the nodes giving highest score when following the $P_T(v)$ policy.

After providing good results in Computational GO[11], MCTS gained a large amount of attention in the research community. Additionally, due to its scalable any-time approach, allowing the algorithm to be stopped at any time, MCTS is still able to output usable solutions or continue simulations to produce better results over time [Coulom, 2006].

The vanilla MCTS can be described as an iterative approach to expand the search tree into directions which prove to be promising. The approach starts with an empty root node containing the current state s_0 and adds new child nodes to transition to new potential states s_i, where i is bound by the number of simulations. The transitions from state to state are actions a, similar to transitions in FSMs. The most promising or "urgent" nodes are then further expanded in an iterative manner, creating an unbalanced search tree. The terms promising and urgent are defined in the approach by an

[11]The boardgame GO is a complex turn-based two player game. It is similar to other classical games such as chess as it requires a player to strategise and plan moves ahead. In contrast to chess, the state space of GO is larger and more complex, making it a challenging domain for artificial intelligence research.

internal criteria for selecting a child. A more detailed description of such a criterion is given further down. The main approach steps as described by Brown and Nee [2012] are illustrated in figure 2-14.

Algorithmic description of MCTS:

1. *Selection:* Traverse the search tree starting at the root to a child node by applying a given policy $P_T(v)$. Stop the traversal once a child node is reached which is either selected for expansion or is a terminal.

2. *Expansion:* Add a new child node, if the current node is not a terminal node. A terminal node in a game setting would finish the game. A child node j represents a new state s_j of the system, achieved by executing an action a_k on it its parent node k.

3. *Simulation:* A simulation is executed using the current state s_j of the system to evaluate the outcome/reward of the currently selected child j. This is done by selecting actions according to the policy $P_D(v)$ until the simulation is finished.

4. *Backpropagation:* The result of the simulation is propagated up to the root following the path taken. On its way up all nodes are adjusted by including the result of the child.

This process yields a result a (the best action to take from s_0) after each iteration. Generally, the more simulations can be run, depending on the quality of the simulation, the better the resulting action. The advantage of MCTS becomes visible when the feature space of possible actions at each state is large. In those cases approaches like MiniMax expand the whole tree which is intractable in most digital games.

As described earlier, there are two policies for selection: either which child to select for traversal, or which move to make once the leaf node in the expanded tree is reached. In general $P_D(v)$ is randomly selecting actions as there are no further evaluations possible. A currently popular policy for $P_T(v)$ is the upper confidence bounds for trees (UCT) which drives the tree expansion towards a partial search which balances tree expansion and local exploitation. UCT is based upon UCB1 [Auer et al., 2002] and is defining a good measure for iterative local search.

$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}} \tag{2.3}$$

UCT is given by $\bar{X}_j \in [0,1]$ the average reward or score for choosing action a_j, n the total number the parent of node j has been visited, n_j the number of times node j has been visited and C_p a constant affection the amount of exploration. A more

detailed discussion on the exploration constant and its implication is given by Kocsis et al. [2006].

Due to its success in GO and its any-time capability GO receives a large amount of attention in the academic community as described by Browne et al. [2012]. This attention and its general performance attracted attention in the games industry and resulting in its application in games with incomplete information such as MAGIC: THE GATHERING [Ward and Cowling, 2009]. The advantage of using MCTS in contrast to other machine learning techniques is that it performs well even when no domain knowledge is available. Nonetheless, if domain knowledge is provided and good heuristic can be produced, the approach should perform exceptionally well because the search space will be explored in a more efficient way. The main difficulty when trying to use search techniques including MCTS in current commercial digital games such as STARCRAFT is, how to identify a good trajectory. MCTS uses simulation to explore the space which needs to be available as part of the commercial game, otherwise it is quite costly. Branavan et al. [2011] present an Monte Carlo search approach using domain knowledge based approach for the relatively old game CIVILISATION II. Thus, exceeding the capability of domain knowledge free MCTS. Their approach incorporates extracted linguistic knowledge from the game manual encoded in an ANN to support the action selection process. However, the game state space is enormous—Branavan et al. [2011] estimate 10^{188} states— and the simulation steps within such a space take a large amount of time. Thus, simulation is in this case extremely expensive. This results in a relatively low simulation count, between 100 and 500 simulations, and an extremely large time gap before making a move. During their experiments it took between 10 and 120 seconds before the approach was able to make a reasonable move. The resulting artificial player was able to win against the included adhoc rule-based AI 78% of the time but an experiment against a human player would have been impossible taking the decision making time into account. Nonetheless, the approach shows potential directions for future work or motivates game developers to include a less costly simulation engine into their games.

A fruitful approach could be in hybrid systems similar to the GOAP approach discussed on page 81 where a re-active planner is combined with A*. If MCTS would be used in plan space to come up with solutions to planning problems, the state space would be more confined and potential domain knowledge could reduce the number of simulation steps, resulting in an AI able to compete with human players.

2.1.4 Summary of Approaches to Game AI

The presented survey of approaches common in games may not have covered *all* existing game AI systems. However, the focus was put on incorporating the most significant and influential ones which are not only frequently used but also represent a good sample of what will be utilised in the foreseeable future. Most of the techniques presented are address lower level and less complex design and implementation decisions, requiring only a shallow learning curve and have a broad well-grounded background literature. Finite state machines are the perfect example because they are the standard modelling approach for game behaviour due to their visual presentation and their simple implementations. More sophisticated methods such as the physics-based Potential Fields are treated as black box add-ins into game systems. They again visualise the decision-making process well and are computationally less expensive, yet robust. BT became popular because it addressed the need for better control over larger behaviour systems, a situation FSMs could not handle any more. They are used more as a framework than an actual tool which results in different tool-kits mostly well integrated into a programmer-oriented toolkit. This results in them also being less design friendly than the FSMs. However, they are a powerful advanced approach to agent design and agent frameworks. Commercial productive systems rarely use evolutionary and learning methods due to their long reasoning times or computational costs on one side. Due to their steeper learning curve and insufficient coverage of new more scalable and robust approaches on the other side, most of the used learning approaches in games resemble vanilla textbook systems. This usage of textbook solutions is due to the divide and difference in the presentation in industrial and academic knowledge exchange when it comes to new concepts and methods. MCTS is a perfect example of a new approach being introduced into games. It not only is scalable but it also is presented in an industry oriented way of the state of the art of academic work.

The next section introduces the term agent and discusses methods and tools which are high-level approaches to agent design, coming either with their preferred design methodology or enforcing it implicitly to generate flexible modular advanced agents. They represent the state of the art work used in games or game-related areas.

2.2 Agents and Agent Design

The previous section discussed underlying concepts essential to building game AI systems. We first focused on decision-making as a substantial part of creating entities acting upon information in an environment. We then examined spatial approaches which offer efficient algorithms that can be used without the need for substantial ad-

justments. Most games require entities to interact in spatial environments; thus, the presented approaches provide a fast integration with stable results when applied. After examining spatial approaches, evolutionary methods were analysed, showing their potential to improve systems but also highlighting the difficulties in using them.

In this section, we will introduce and discuss the term agents and its meaning in game environments. We will look at different architectures and design methodologies for agents. Thereby we take existing analyses of agent architectures done by Laugier and Fraichard [2001] Bryson [2000a] on decision systems and robotic architectures and Langley et al. [2009] on cognitive architecture into account. We specifically focus on architectures which can be applied to digital games.

To get a better understanding of the different perspectives in building agent systems and their Decision-Making System (dms), this section contains an important separation between approaches which would be classified as light-weight cognitive systems and systems which are classified as fully cognitive. From Section 2.2.2 to Section 2.2.4, the presented approaches are more driven by the motivation of generating results and building applicable working artefacts. Sections 2.2.5 to 2.2.8 approach agents from a more cognitive perspective, modelling the underlying system and reasoning. They are driven by the idea to understand and model human or animal cognition in the pursuit of knowledge about them. By doing so interesting results which are also applicable to IVAs emerge.

> "Psychology has arrived at the possibility of unified theories of cognition—theories that gain their power by positing a single system of mechanisms that operate together to produce the full range of human cognition.
>
> I do no say they are here. But they are within reach and we should strive to attain them." [Newell, 1994, p.1]

2.2.1 Agent Design

What are agents? How do agents differ from game characters? To understand agents, we start off with a broad definition and refine to a point suitable for the purpose of this work. Defined in the broadest way, agents are components acting or re-acting upon an environment. Thus, agents can range from chemical agents to robots to human beings. A better distinction is offered by introducing Natural Agents and Artificial Agents. The first term refers to living organisms such as bacteria or animals, including humans. The second term refers to acting constructs which have been created, e.g. chemicals, robots or acting software components. In games, Artificial Agents range from non-player characters (NPCs) to the sub-system controlling the

progression of the game.

The most important point when only focusing on games and software systems is that the broader term Agents is uniting the player/user, the non-player characters and the software under its definition. Both agent types can act within the game environment based on its current state. Other definitions, such as the one by Russell and Norvig [1995], go further into specific directions, detailing in greater depth artificial agents. A notable differentiation they make is between different levels of activeness and autonomy of an agent. In our work, it currently suffices only to differentiate between natural agents—e.g. the player—and artificial agents—either systems controlling one or more characters or systems in charge of controlling the environment or the flow of the game.

Game characters, in contrast to game agents, do not necessarily classify as individual agents. Game characters are animate objects in game environments sharing shapes similar to those of natural agents such as animals. They are similar to game pieces in chess and other board games where all the agency and action-selection capability is with the player or the controlling agent and not with the actual game piece. However, some characters, namely non-player characters (NPCs) are portraying the impression of agency and are in some cases embodied artificial agents. The differentiation between agent and character is important when analysing and developing games. Agent design in most cases relates to the reasoning and action selection whereas character design is mostly referring to graphical representations of entities.

Burke et al. [2001] position agents in the centre of attention for their work on game experience by defining Deeper Gaming Experience. To understand its relation to agents better, we will use a modified version—deeper agent behaviour—which focuses on situatedness, reactivity, expressiveness, soundness and scalability of the agent. Each of the elements can be seen as dimensions which form a space for agents to be placed in.

Based on our definition of agents, we are now able to discuss the process of modelling or designing artificial agents. For now, we leave the discussion of designing *for* natural agents aside and focus on artificial agents. Designing an agent for early games such as Space Invaders required only the creation of a two-dimensional shape which moves vertically or horizontally across the screen. Due to the limitations of the hardware and systems at that time, more sophisticated approaches or agents were not feasible. The availability of more storage and computational power lifted this restriction as it became possible to use simple finite state machines (FSMs). FSMs offer a way to model an agent that has multiple states and transitions to get from one state to another. This approach was widely used as it is thought to be intuitive [Raskin, 1994] for humans to break down problems into different states and solve them individually by creating

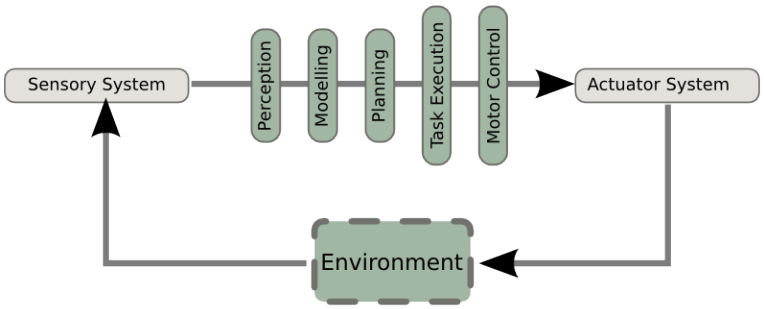


Figure 2-15: A traditional decomposition of robot and single agent systems into functional models presented by Brooks [1991]. Game agents, in this respect, are highly similar to robotic systems. This is visible when taking a closer look at the c4 architecture, see Section 2.2.7.

transitions.

To create DEEPER AGENT BEHAVIOUR, programmers moved to more sophisticated techniques such as the discussed BT. It took until the early 2000's for the game industry to be able to step beyond state machines and include planning approaches and more cognitive science inspired approaches. In contrast to that, robotics research was able to cross those boundaries decades earlier with planning systems such as the PROCEDURAL REASONING SYSTEM (PRS) [Georgeff and Lansky, 1987] or the SUBSUMPTION architecture, which will be described further down. With the growing complexity of tasks in less and less restrained spaces, robot reasoning and action selection had to be fast enough to drive and interact in a reasonable time.

To build robotic systems, as a first step, the roboticists assessed the behaviours the robots had to perform. From those functional requirements, modules were designed. Those modules formed first drafts of the new system. In figure 2-15, a traditional robotic system approach is presented. The signal flow and reasoning process before making a move was relatively long and could take up several minutes as stated by Mali [2002]. To introduce shorter response times, reactive systems were introduced, able to tackle this issue. Games in this respect are highly similar. Game agents are in most cases acting entities within a given, mostly non-static environment. They underlie strong resource restrictions and require sophisticated action-selection mechanisms.

The following approaches present architectures and methodologies which can be used to support the creation of game agents without falling into the same pits as early robotics projects. The c4 system, discussed in Section 2.2.7 builds upon early robotics research, building an underlying functional structure similar to the one presented in figure 2-15. From this traditional architectural view the claim to have shifted their

system towards a Behaviour-Based AI approach, an approach for reactive robotic architectures. Magerko et al. [2004] uses the Soar architecture, which will be described later on, to build cognitive game agents within a virtual environment focusing on modelling cognitive agents within a game environment.

Belief-Desire-Intentions BDI [Rao and Georgeff, 1991] based agent architectures are many and are centred around declarative descriptions of single or multi-agent systems [Wooldridge et al., 1995]. They and expert systems have been omitted from this work as they focus on a declarative problem description which has not seen applications in commercial games. However, most of the systems described further down allow the modelling of agents based on the three main attributes of BDI. The concept of using beliefs, desires and intentions has been applied numerous time loosely within most game-based systems. In those loose implementations, beliefs represent facts about the state of the environment and the agent. Desires represent goals the agent wants to achieve and intentions are operators/rules applied to the state of the world to achieve goals. By considering the importance of all three attributes, the design process for agents can enrich the resulting system as discussed by Rao and Georgeff [1991].

2.2.2 Behaviour-Based Artificial Intelligence

Behaviour-Based AI (BBAI) or to be more precise Behaviour-based robotics was first described by Brooks [1991] to describe computational models which produce direct behaviour, such as move(x) which results in a spatial movement. Brooks [1991] identifies following key characteristics which describe Behaviour-based approaches for robots:

- The robot is situated in a physical environment and has to interact directly with it.
- The robot is embodied. Thus, it is part of the dynamic system forming the environment. The robot senses through its body and acts with it.
- The robot is judged based on its performance within the environment and through interactions in terms of its "intelligence".
- The "intelligence" of the robot is an emergent property of its actions and how they are perceived.

BBAI as introduced by Brooks [1986] adds modularity to a FSM system. Multiple finite state machines are layered, each specialised to a sub task of the problem. Thus, the resulting agent is easier to develop and program as each layer can be worked on

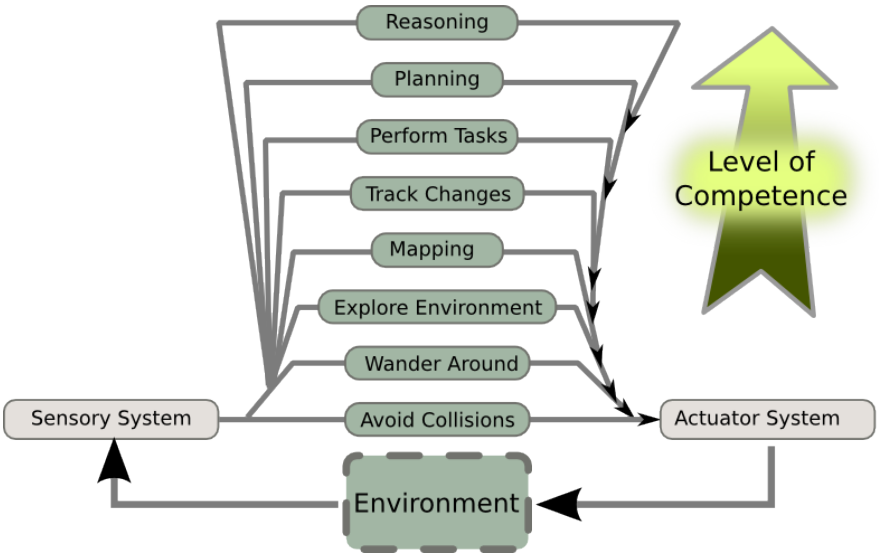


Figure 2-16: The SUBSUMPTION architecture presents an alternative approach to agent architectures. In contrast to traditional design, see figure 2-15, an agent controlled by the SUBSUMPTION architecture has access to all sensory information and uses layered control between modules of lower and higher competence.

separately. In the original BBAI methodology, the *Subsumption architecture*, FSMs can only have interactions with one another by modifying the inputs and outputs of other connected or linked FSM/modules, see figure 2-16. Different layers of control are built into a simple stack hierarchy, where the higher layers can subsume the roles of the lower level modules by modifying their outputs appropriately. This is the main idea behind SUBSUMPTION, an AI is built bottom-up with the overall goal on the highest layer of the system. The original proposition of BBAI made no inclusions for machine learning, memory or planning. In this form it is a purely reactive architecture, however one more robust and extendable compared to a basic FSM as discussed by Laugier and Fraichard [2001]. Later BBAI approaches used representations other than FSMs for the different modules within the SUBSUMPTION hierarchy. Those modules include approaches such as potential fields [Konolige and Myers, 1998; Arkin, 1998], planners [Burke et al., 2001; Bryson and Stein, 2001] or learning mechanisms [Isla et al., 2001].

Mali [2002] takes a different approach to BBAI, he abstracts the definition, rephrasing and extending the four main characteristics. His points are also more in line with behaviour-based software agents. He states that an agent is situated in an environment and is able to perceive it, similar to Brooks first point. Instead of being embodied, the agent posses resources of its own. Those resources in terms of a robot can be an embodiment but it can also refer to an encapsulated memory space or sensors which are not

shared with other agents. Instead of focusing on the emergence of intelligence through the agents actions, Mali focuses on the emergence of useful behaviour to the agent. Similar to Brooks, Mali does not support the idea of internal memory beyond state information as useful or feasible for action selection.

Having no internal representation of the environment, Mali [2002] argues for the usage of environment markers and behaviour coupling to reduce computational costs. In nature, marker based approaches also exist. However, they normally go beyond the simple purpose of outsourcing computation of locations. Territorial markers are also a means of communication between different agents and carry extra information which can be extracted from them. Additionally, the approach that Mali [2002] proposes requires intervention from the environment of other agents to set up or move the marker. Because of this, the approach seems not feasible in non-closed environments where additional set up is required. The second proposal is the usage of behaviour coupling which is similar to joint behaviours introduced by ABL in Section 2.2.8. To restrict the amount of search in design space and to chain meaningful behaviour, stimuli are coupled. The approach would be similar to matching preconditions in a planner to trigger sequences. In BT a similar mechanism is used which is the sequence node, see section 2.1.1.

2.2.3 Goal-Driven Autonomy (GDA)

Muñoz-Avila et al. [2010] demonstrate the extension of online planning to the domain of games. In contrast to classical planning, GDA introduces a way of handling unexpected events during the execution of a plan [Molineaux et al., 2010], thus, allowing the planner to shift attention from one goal to another. It also allowed for re-planning of actions to pursue the current goal. The approach consists of five different components illustrated in figure 2-17. The Environment Σ allows a way to derive observations about itself. This includes changes to its conditions and states. $|Sigma$ contains its current state s_c as well as its previous states. The controller, upon receiving a new plan p, performs the contained actions on the environment. The performed action is able to alter the environment visible in future observations. The model of the environment M_Σ in conjunction with the state s_c and the current goal g_c is presented by the controller C to the planner. The planner based on this new information generates a new action plan. The plan p consists of a pair containing a set of actions A, which are to be executed, and expectations about the state of the environment X. The controller has also access to goals G which form a set of all currently pending goals, including the current goal g_c. Upon receiving new observations s_c, the controller uses the Discrepancy Detector to evaluate the expectations about the environment against the current

observations. If discrepancies d are detected, the Explanation Generator hypothesises on them and proposes an explanation e to the Goal Formulator. The Goal Generator, at that point, creates a new goal g based on the discrepancies and the explanation for the Goal Manager. Receiving a new goal g, the Goal Manager selects from the set of pending goals the one to pursue next. This approach enhances existing reactive planning approaches as discussed by Ghallab et al. [2004]. It offers a way to shift attention from one goal to another on an event basis instead of either waiting for a goal to be finished or re-planned.

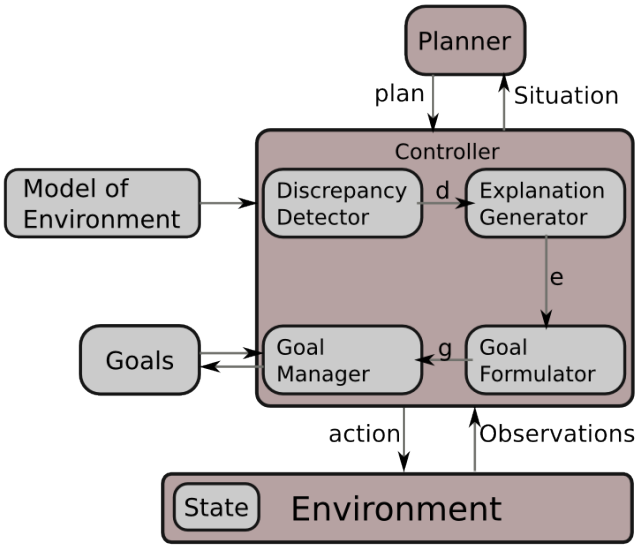


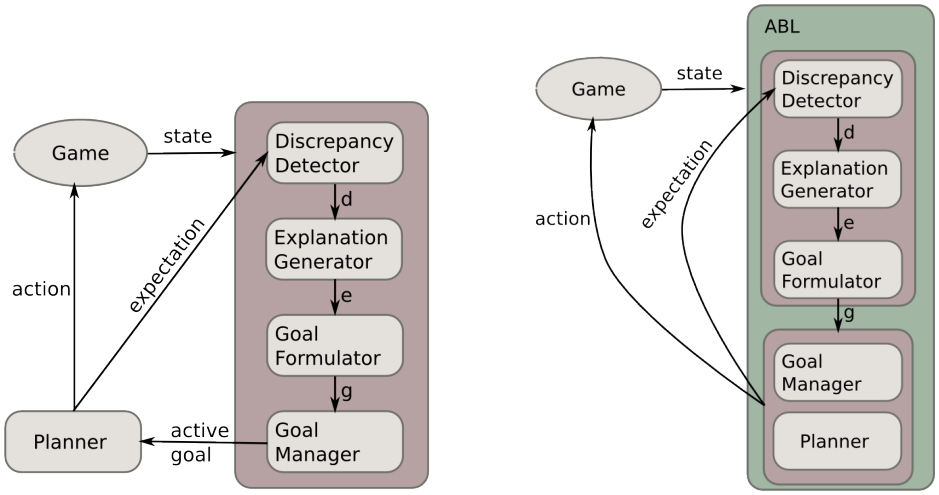
Figure 2-17: The Goal-Driven Autonomy model based on Molineaux et al. [2010] illustrates five major components: the Environment Σ, the Model of the Environment M_Σ, the planner, the goal list G, and the controller C.

Muñoz-Avila et al. [2010] compare their planning approach to the dominant approach in Games, BT [Champandard, 2007d] but make an incorrect assumption about their capabilities by stating:

> "Research on game AI takes a different approach to goal formulation in which specific states lead directly to behaviors (i.e., sequences of actions). This approach is implemented using behavior trees, which are prioritized topological goal structures that have been used in HALO 2 and other high profile games (Champandard, 2007). Behavior trees, which are restricted to fully observable environments, require substantial domain engineering to anticipate all events. GDA can be applied to partially observable environments by using explanations that provide additional context for goal formulation." [Muñoz-Avila et al., 2010, p.466]

As presented in Section 2.1.1, BT does not make mention of requiring fully observable environments. Game developers often relax the condition to reduce computation time but the approach itself does not require a fully observable environment to be implemented. To create a BT the agent utilising it only needs to access given information to plan actions. The BT is agnostic to which information is accessible. The statement about substantial domain engineering however is partially true but applies to GDA as well as presented next in the approach by Weber et al. [2010a].

Supporting the idea of a goal-driven approach, Weber et al. [2010a] developed an agent system—EISBOT—implementing the GDA concept for STARCRAFT. They aimed for an agent that could reason about its goals and anticipate game events instead of purely reactive systems which only respond to changes instead of anticipating the future by maintaining a model. To build their agent, they started with a simplified the GDA model to better fit their purpose, see figure 2-18a and used the ABL system, see Section 2.2.8, to build the agent.



(a) Components of the simplified Goal-Driven Autonomy conceptual model based on Weber et al. [2010a]

(b) EISBOT GDA architecture using ABL for StarCraft.

Figure 2-18: (a) presents a simplified version of GDA for games incorporating the environment and its model into one component. (b) shows the EISBOT implementation of the GDA adapting the GDA to fit in line with ABL.

The model they present differs slightly from the original model presented by Molineaux et al. [2010]. The planner and Goal Manager are merged into one component. This component receives the goals to activate from the Goal Formulator and also maintains pending goals. This differs from the original model as they clearly separated the two components. The original model also allowed for a separation of data and imple-

mentation. In contrast to this, the created system presents an interleaved architecture not separating between them. The planner generates actions for the environment, in this case the game StarCraft. It also generates action-correlated expectations about their impact. Those are given to the Discrepancy Detector. Upon receiving a state update, the detector then, as described by Muñoz-Avila et al. [2010], checks for diverging expectations. The derived discrepancies are used by the Explanation Generator which is a simple hand-crafted "if-then" block. This reduces the hypothesis building to a mere check. However, this approach still allows the anticipation of future events which supports the need for expert designers. The Goal Generator uses the explanation either by using again a hand-crafted case block or by selecting among several goals and forwarding them to the Goal Manager. The manager, as discussed earlier, is part of the planner and activates all new goals.

In contrast to the work of Muñoz-Avila et al. [2010], EisBot allows parallel execution and pursuit of more than one goal. The manager part of it is additionally divided into different sub-groups within the Abl system. They independently track and pursue goals based on the existing game tasks. The EisBot is also heavily dependent of domain engineering which according to Muñoz-Avila et al. [2010] is not beneficial for planning systems. They are putting it on similar terms to Bt or other game related approaches. Nonetheless, EisBot presents a strong approach which can compete with the original AI implementation of the game as well as moderate human players, as illustrated in their work. The goal-driven autonomy components within EisBot are purely implemented in Abl as sub-trees. This makes it hard to extract or replace them. However, this deep integrating results, as discusses earlier, in a system able to compete with average human players, yet maintains an amount of modularity and flexibility outperforming the original agent within the game.

2.2.4 Goal Oriented Planning (GOAP)

Goal-Oriented Planning (Goap) was introduced by Orkin [2004] to address the challenges in games which outgrow the capabilities of earlier techniques such as finite state machines or ad hoc rule systems. Orkin argues for a real-time planning system for games which uses a symbolic representation of strategies and focuses on goals. In his introduction, he states that with each new game generation the bar for AI in games will be raised. However, looking at commercial games reveals that this does not seem to be the case. AI is an important part of games and a game can benefit a lot from the underlying AI system. Nonetheless, games such as Heavy Rain or World of Warcraft are not measured by their AI capabilities but by other factors.

Goap features a regressive real-time planning approach built on top of the Planning

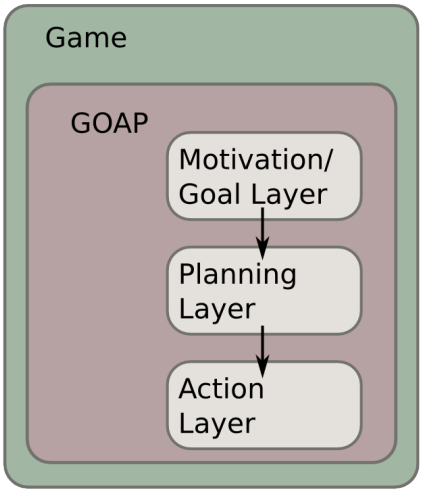


Figure 2-19: The GOAP architecture representing three distinct layers based on their main purpose.

Domain Definition Language (PDDL). It provides a modular architecture to share behaviours among NPCs. The goals and actions of the system are atomic, making them easier to maintain and allow layering and sequencing. A strong point GOAP makes is the separation of implementation and data to support the workflow of teams working on shared tasks. Thus, the planning layer and the underlying game implementation do not mix. Based on its planning origins there is also no explicit link between actions and goals. During plan time a goal is selected to be pursued and the planner chains actions based on their preconditions and post-conditions to arrive from the current state at the goal. GOAP can be considered as a three layer system, see figure 2-19.

The intended workflow to foster behaviour sharing and better development is a two step process:

- Engineers/Programmers develop a pool of actions and goals. This includes the specification of their preconditions and post-conditions/effects which are used by the planner to derive working plans at run-time. The actions and goals are atomic and encapsulated and dependencies between them are specified in the pre-conditions.

- Designers are able to assign actions and goals to specific agents using a data file. They are not able to modify the two atomic types or influence the planner in any other way.

The driving force behind this shared workflow is that designers should not handle the micro-management of behaviours. They are supposed to concentrate on the high-

level design using goals. To be able to react to the game environment its state is encoded in a fixed size array containing symbolic representations of the most important properties, for example locations where players are. During planning time, the planner then searches the game state and action pool. It is hashing actions with its effect type to reduce the search space. The search is carried out using A* which was introduced in Section 2.1.2. As planning and search are quite costly a plan is only re-planned if it has been invalidated or if the currently relevant goal changes. A limitation of the system discussed by Orkin [2004] is the fixed symbol space used for planning. This issue is revisited by Orkin [2005].

In the later work, Orkin [2005] demonstrates the application of GOAP to real-time, first-person games in the commercial product F.E.A.R.. According to the author, the reasoning part of the AI resembles the c4 system, discussed in Section 2.2.7. In contrast to c4, the action tuples of c4 are replaced with the previously mentioned planning system. The planner plans for individual agents but allows only one active goal at a time. It re-plans and re-considers the pursuit of the current goal after "sensors detect significant changes in the state of the world" [Orkin, 2005, p.106].

A further elaboration of what a significant change is, is not given and is most likely case dependent and hand-tuned. Another important advancement is the usage of a blackboard and working-memory facts (WoMe).

A working memory fact contains a fixed set of attributes which are in turn associated with a confidence value. The usage of the confidence varies depending on the attribute. This varying usage makes the design less intuitive. Nonetheless, the confidence relates to what priority or accuracy an related attribute has. The WoMEs are centrally stored on the blackboard. This allows the agent to access them at any time, a feature FSMs are missing.

The blackboard is used to decouple the execution of the game and the planner/reasoner. To execute an action a related blackboard symbol is set. To retrieve sensory information the blackboard is checked without querying the game. This decoupling allows the system to schedule heavy tasks to balance the load on the CPU. Expensive computational task such as a sensor using raycasts are now amortising over time as they are not called every frame or can occupy resources for multiple frames before their result is needed. Nonetheless, this introduces inaccurate or outdated information into the reasoning process so the timing of updates impacts not only the CPU time but also the expressed performance of the system. Comparing this approach with animal sensing however highlights similarities. Biological senses update also at different times and sometimes not instantaneous. Below 50ms even trained human ears have difficulties detecting flaws in audio streams. With current games the system generally runs at 60

frames per second, allowing the system to update every 16.7ms. Thus, a human ear cannot differentiate if there is a pause for 3 frames. However, timing and scheduling a multitude of updates to spread CPU cost is considerably difficult and an ongoing research area.

Orkin [2005] extends his initial GOAP approach and optimises it within the game F.E.A.R. Thus, demonstrating a reactive goal-oriented planning approach within a commercial product. He argues that *in-game* approaches are only beneficial if they bring a noticeable improvement. Due to the planner and the large search space, the team saw one of those improvements, the emergence of game behaviour which was not programmed but created as agents re-planned actions. This and the ability to react to unforeseen events make planning approaches interesting for games.

2.2.5 Heavy Cognitive Architectures

Based on the study of human cognition and directed towards unified theories of cognition [Newell, 1994], multiple cognitive architectures emerged since the 1970s. The two most prominent ones are SOAR and the ACT-R architecture. Both theories are based on production rules, implement short-term and long-term memory and learning. In chapter 5, a more light-weight approach to creating agents is presented which is heavily inspired by the underlying mechanisms of the presented heavy cognitive architectures. However, this new approach tries to remain as light-weight as possible to still be applicable to games or resource restricted environments. A strong influence of cognitive approaches is also visible in chapter 182 where a novel augmentation to augmenting selection process is presented, based on mechanisms in the mammalian brain.

Adaptive Control of Thought-Rational (ACT-R)

ACT-R [Anderson, 1993], in contrast to the later described SOAR, has seen no implementation within a digital game system. Nonetheless, the system is highly similar to SOAR, prominent in the research literature and well known for modelling human behaviour and human task learning.

ACT-R is organised into a set of modules each responsible for different types of information. Figure 2-20 presents a high-level view of the ACT-R system. The system contains modules which are in charge of acting, sensing, goals and declarative long-term knowledge. Additionally, each of those modules has an associated buffer representing the state of the module. Those buffers form the system's short-term memory and contain declarative knowledge called chunks. Chunks have a type and contain a value

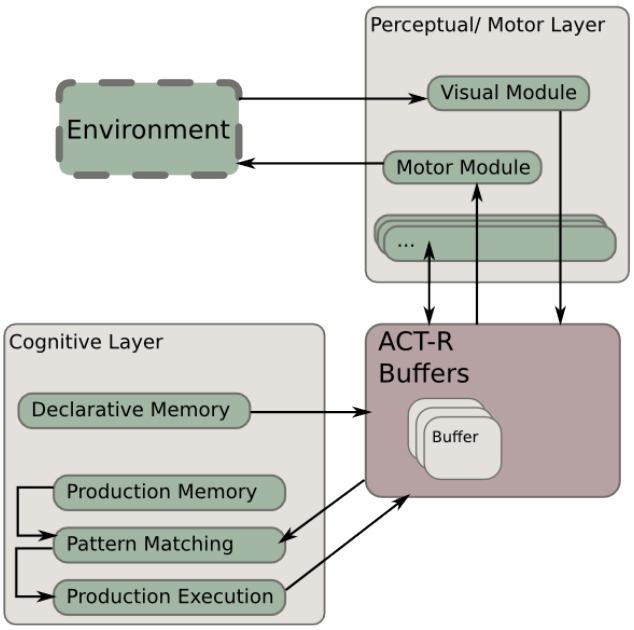


Figure 2-20: The ACT-R system contains modules which handle different types of information Anderson et al. [2004]. The buffer is the interface between different modules and represents the short-term memory of the system.

defined by the type. Their purpose is to communicate information between different modules through the buffer. The ACT-R system operates by matching productions against the current short-term memory buffer. Each production has an associated base activation which is determined by its past usage. The base activation is a key factor determining which production will be loaded form memory. So, each cycle ACT-R searches for productions matching the buffers and selects those with the highest utility. The utility is calculated the cost of loading a production plus its benefit. The benefit of a production is a numerical value of how desirable the production is, multiplied with its chance of success, plus some added noise. This process of matching, selecting and executing productions is continuously repeated whereas the goal is that each cycle only takes 50ms [Anderson et al., 2004, p.1048].

The system is able to learn by updating the base activation of productions at each cycle. Thus, productions which are frequently useful are more likely to have a high utility. The probability of success is updated whenever the corresponding production is used. This allows for a more correct long-term evaluation not taking drastic changes in the environment into account. New production can also be learnt through production compilation. A process which combines firing rules and replaces constants which match a solution with variables. A more detailed description of the process is given by

Anderson et al. [2004].

State, Operator and Result (SOAR)

SOAR is the second full cognitive system which we will have a closer look at. The system can be divided into four major parts as presented in figure 2-21. Similar to ACT-R, it is based on production systems and contains short-term and long-term memory and learning. SOAR contains three types of knowledge:

a Operators store long-term knowledge as production rules. The operators are organised to cover the current problem space.

b Episodic knowledge is stored in snap-shots of previous states of the working memory.

c Semantic knowledge stores factual information in the form of individual elements of working memory for later retrieval.

Productions within SOAR are organised as tasks to satisfy goals. Tasks are contained in the short-term memory and combine operators and goals. This process is different from ACT-R where productions are purely selected by their utility and matching the current state. SOAR uses a problem state and an initial set of operators to traverse the space from initial state to goal state. From the presented knowledge types, types (b) and (c) are relatively new to SOAR, allowing the system to learn factual information which is not in the form of production rules but complex situations in the form of memory and perceptual snap-shots. Those forms of memory are not present in all versions of SOAR but represent a way to learn declarative knowledge. They additionally are quite memory intensive because the system continuously learns whenever encountering unknown situations.

SOAR approaches a goal within a selected problem space by proposing, selecting and applying operators to the current states. This is done by the decision system, see figure 2-21. Because the operators are goal-directed, they perform deliberative acts aiming to fulfil the agents goals. If no production can be used to fulfil the current goal, based on the agent's knowledge, an *Impasse* happens.

Impasses are one of the ways the system learns new productions through chunking[12]. Chunking takes the current state and creates a new intermediate goal to solve the impasse. For this, the system opens a new problem space and initiates an initial

[12]Chunking in SOAR is different from ACT-R chunks which is generating confusing statements while comparing both systems. However, the process is quite different as ACT-R chunks are declarative or procedural knowledge but SOAR's chunking generates operators within long-term memory to satisfy a goal.

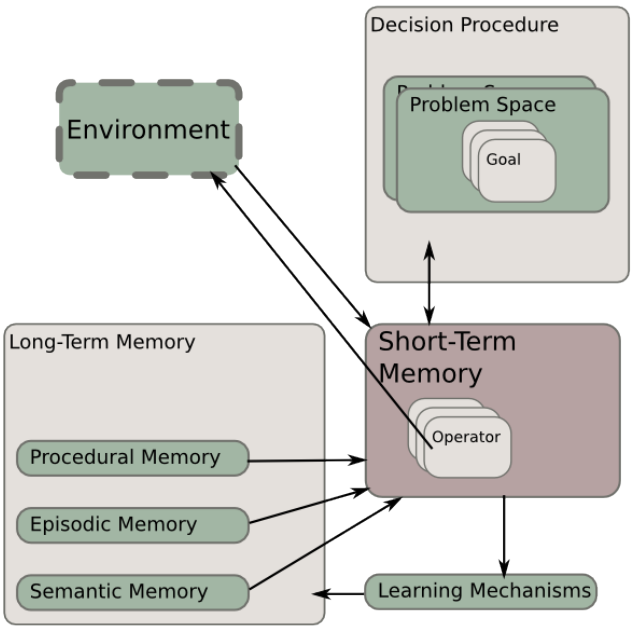


Figure 2-21: SOAR is organised around the short-term memory which is used to apply operators to the current state of the world. It has an independent decision making system proposing and selecting operators fitting the current goal.

problem space internal state that is based on the current super-state of the original problem space. From this state, the DMS applies a production altering the state towards the new goal. If the impasse is solved by arriving at the goal state, a new operator will be added by combining the productions from within the Impasse problem space based on their trajectory within the space. If the goal state is not reached, the process continues recursively opening a hierarchy of sub-goals until the impasse is solved. This mechanism of Chunking is similar to planning approaches discussed by Ghallab et al. [2004]. A more in-depth description of the process is given by Laird et al. [1986]. Similar to planning, the Chunking process is quite costly however once a new operator is learned the cost can amortise over time.

In contrast to ACT-R, SOAR has been used frequently to model and design agents [Wintermute et al., 2007; Van Lent et al., 1999] and synthetic characters [Laird et al., 2000; Magerko et al., 2004; Assanie, 2002].

Laird et al. [2000] present a game architecture for creating narrative-driven adventure games. The environment uses UNREAL TOURNAMENT in combination with SOAR to develop a game test-bed going beyond the first person shooter agents developed in earlier projects [Laird, 2001]. Laird integrates SOAR through an abstract interface to the game and allows the development of a large scale of agents within it. Magerko

et al. [2004] presents an AI director which augments the decision process of a story. The approach is similar to the story beat system presented by Mateas and Stern [2002] but focuses on a fixed hand-authored story which is experienced in a defined sequential order. Even so the agents are relatively light-weight, the system is able to run over ten agents at a frame rate of 30 Frames per Second (fps) on a single machine which creates a character rich environment.

2.2.6 ICARUS

Productions systems are the foundation for Icarus as well as SOAR and ACT-R and they are a uniting feature for most other full cognitive architectures. Langley et al. [1991] introduce the architecture and underlying theory of cognition for ICARUS with the aim of creating a unified theory of cognition [Newell, 1994]. Instead of using chunks or operators, ICARUS uses concepts and skills to drive the system and the controlled cognitive agent [Choi et al., 2004]. The system provides short and long-term memory as well as a learning mechanisms for new skills and concepts. *Concepts* form the first part of reasoning and describe environmental situations by either referencing other *concepts*, or by taking perceptual information acquired by the system into account. *Skills* as the counterpart specify how to achieve goals set by the system. They can be achieved by decomposing them into sub-goals until primitive actions are reached within the goal hierarchy. ICARUS uses hierarchies for concepts and skills to create complex behaviour. Both, skills and concepts work hand in hand to approach cognitive tasks by splitting information, similar to SOAR and ACT-R, into declarative and procedural knowledge.

The system operates within an environment by interpreting perceptual information and storing those interpretations as descriptions in the short-term memory. The descriptions represent beliefs about said environment. Figure 2-22 illustrates an interpretation of the ICARUS architecture based on Langley et al. [1991]; Choi et al. [2004]. The ICARUS system consists of four main components:

- Labyrinth is responsible for storing and accessing the long-term memory.
- Argus is the perceptual module which contains one part of the short-term memory related to sensory information perceived by the agent and the current beliefs.
- Mæander connects the agent to the environment through interaction. It controls and trigger motor modules and it contains the goal buffer which drives the production rules within the skills.

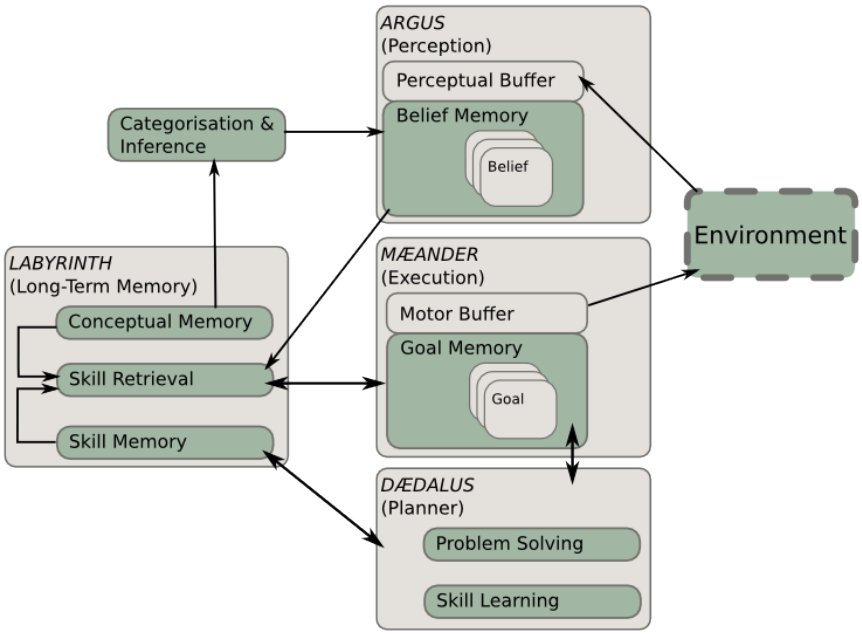


Figure 2-22: The ICARUS architecture can be represented through four different modules. The long-term memory—Labyrinth —stores both conceptual and skill memory and the skill retrieval is responsible for loading skills into the short-term memory. The short-term memory in ICARUS, situated in Argus and Mæander. It consists of the goal memory which is connected to the motor buffer and the belief memory connected to the perceptual buffer.

- Dædalus generates plans by connecting skills to satisfy goals. It is also responsible for learning new skills by connecting lower-level skills and building up hierarchies to achieve a goal.

The cognitive system uses its internal goals to trigger motor behaviour which affects the environment. The updates of both buffers form an iterative cycle allowing the agent to alter the beliefs and act based on them. Thus, the system drives the agent goals which are motivated by the currently active beliefs. Goals are satisfied by loading appropriate skills from long-term memory. Those skills are loaded based on their preconditions which are in turn sets of beliefs about the environment and state of the agent. By interpreting perceptual information, the system finds matching primitive concepts in the long-term concept buffer which may be useful. Initially, all affected primitive concepts are loaded based on the system perception. Lower-level concepts trigger the loading mechanism of higher level concepts. This process creates a logical chain that can form a concept hierarchy because all connected children are also loaded. Once the appropriate high-level concepts are loaded satisfying the goal, the skills preconditions

are satisfied.

If no path from the current state towards the goal state can be found, a backward chaining search within the skill buffer is conducted which interleaves skill execution and search. This process is costly but reduces reaction time compared to a full plan creation. To make ICARUS more reactive, the system disconnects plan execution done in MÆANDER and planning done in DÆDALUS, making the update-action cycle asynchronous. An additional difference to SOAR and ACT-R is the continuous update of concepts whenever they are retrieved. This feature, on one hand, is more time consuming compared to a separate update and retrieval used in ACT-R or SOAR. On the other hand, it introduces a hill-climbing approach to constant adaptation in the environment making the approach more human-like, as argued by Langley et al. [1991].

The agent Choi et al. [2007] developed to demonstrates the capabilities of ICARUS. It is situated within URBAN COMBAT[13] and offer a real-time simulation in a multi-agent environment which allows players to join as well. To reduce the engineering burden object recognition and spatial location matching was reduced from a 3D space to region maps, similar to the previously discussed NavMeshs in Section 2.1.2. Instead of focusing on learning through numeric adaptation of parameters, Choi et al. [2007] concentrate on learning skills through abstracting agent actions and matching those to a template agent behaviour model which represents template skills. Rather than focusing on human-believable agents which have to be evaluated through user testing and qualitative studies, they focused on agents learning different settings and task as the first measure of success for their system. This presents a sensible first step as it shows the functional soundness of the system. They discuss one flaw of their approach briefly, the location memory in their URBAN COMBAT agent does not degrade or build up over time and is perfectly transferred between individual runs of the environment. In contrast to that, the process of human spatial learning is different, it involves repetition and reinforcement of knowledge before remembering. The architecture provides the process needed to address this issue. It would only require modifying the existing agent which involves more domain specific engineering on the agent side. One point which is not mentioned is how well ICARUS scales to more complex scenarios or agent numbers, a shortcoming which has not been addressed even in more recent work.

2.2.7 MIT cX agent architecture

The synthetic characters group at MIT developed the cX system for controlling virtual characters in game environments as part of their agenda to understand and model

[13] URBAN COMBAT is a simulation environment based on the commercial game Quake3 and available at `http://ailab.wsu.edu/uct/`.

synthetic characters. The most prominent version discussed by Isla et al. [2001]; Burke et al. [2001] is version 4—or c4. c4 aims to integrate cognitive, philosophical and agent research into a single cognitive architecture that is similar to previously discussed full cognitive architectures. It is based on the reactive SUBSUMPTION architecture which Brooks [1991] introduced, but adheres more to the ideas of behaviour-based AI [Maes, 1993; Brooks, 1986]. The reactive architecture approach allows for fast response times and a generally low computational cost of O(n2) [Veres et al., 2011] whereas BBAI provides a way of structuring the system. The c4 system also integrates working memory by taking ideas from Rosenbloom et al. [1993] and a prioritisation of tasks. The main motivation for developing C4 was to create a layered brain architecture for experimenting in virtual environments which is able to control a "reasonable amount" of autonomous creatures at a near-real-time rate[14].

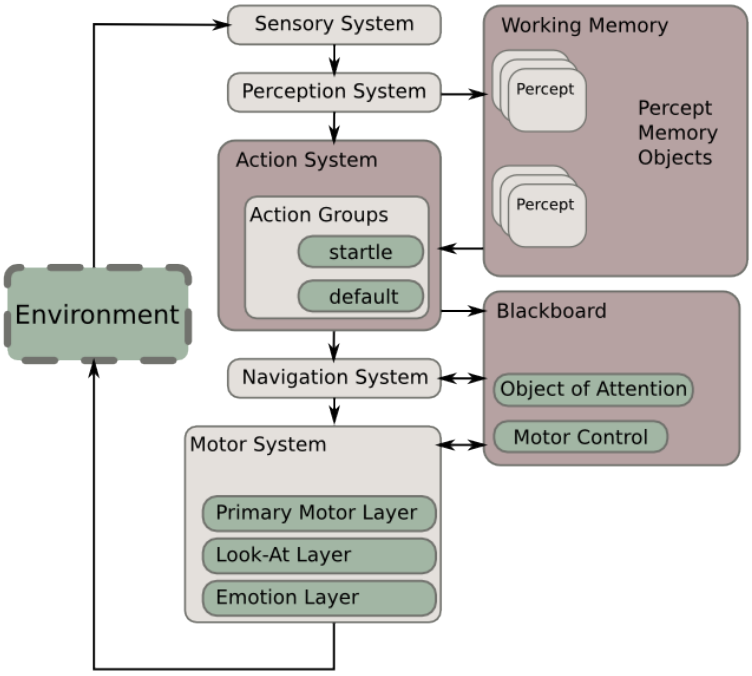


Figure 2-23: The c4 architecture for game environments as described by Burke et al. [2001]. c4 contains five layered modules which built upon each other in a similar way as the SUBSUMPTION architecture by Brooks [1986].

Figure 2-23 visualises the architecture of c4. It presents the five main modules which form the system. The modules represent a hierarchical information or signal processing cycle. The system is implemented by applying a SUBSUMPTION style paradigm using

[14]The systems aimed to run at a 20Hz frame rate which aimed at a less fluent visual representation but introduced an industry comparative strict guideline.

high-level modules for reasoning which override decision from low-level modules. Isla et al. [2001]; Burke et al. [2001] also introduce specialised lower level behaviours for certain tasks such as specific motor control. Those specialised behaviours are called at a later stage, shortly before the execution of an action, able to override high-level decisions. This is done within the navigation system which allows adapting and fine tuning the outgoing behaviour. Burke et al. [2001] introduce this approach as SUPERSUMPTION. It is meant to allow lower level systems to override higher level ones. Their approach is defined as an inverse approach to SUBSUMPTION as defined by Brooks [1986]. However in the SUBSUMPTION architecture, more advanced or complex behaviours are introduced to override lower level and simpler behaviours. Thus, presenting an iterative process towards more robust and complex agents. This means that the SUPERSUMPTION is still aligned with the original SUBSUMPTION paradigm as specialised lower level behaviours are adding only another layer on top of the decision process. Brooks argues for higher complexity and specialised behaviours to override the decisions made by lower-level ones which result in a layering approach to increase functionality. It is an approach sorted by layer complexity. Thus, their definition of SUPERSUMPTION is equivalent to SUBSUMPTION.

The c4 system, similar to current game AI systems or other cognitive architectures, uses a blackboard for creating an asynchronous communication and general purpose storage between the lower level layers of the system which include the navigation and the higher level reasoning. Nonetheless, the blackboard introduces complex dynamics and it becomes harder to track and debug the information flow.

One of the major features the c4 introduces is a differentiation between sensing and perceiving the environment which Burke et al. [2001] claims is inspired by nature. This is based on their concept of PERCEPTUAL HONESTY. In virtual environments, such as games, agents are able to sense or access all information in a non-noisy way. By introducing their sensory and perception system, c4 receives limitations simulating natural environments. Thus, even if c4 is sensing a change in the environment, the controlled agent might not perceive this change due to the perception layer, producing perceptual "honest" sensory information.

Additionally, perceived data is represented as a PERCEPT MEMORY OBJECT (PMO) which can contain different perceived information about the same original data. The percepts contained in a PMO are structured in an object oriented way using inheritance and contain an additional probability value. The value specifies how likely it is that the perceived data is represented by that percept. The combination of probability and inheritance from OOD allow for a complex and sophisticated but computationally expensive perception architecture.

Upon perceiving information about the world the system uses the action selection mechanism and the working memory to execute actions. Actions in c4 are 5-Tuples and an action A is represented by

$$A = [a, tr_c, o_c, s_c, iv] \tag{2.4}$$

where

- **PrimitiveAction (a):** Determines which action should be activated by writing it to the blackboard.
- **TriggerContext (tr_c):** Triggers reference percepts in memory and a relevance correlated to triggering the action.
- **ObjectContext (o_c):**On which object should the action be performed. The object context related to perceptual memory and the attention regarding a specific entity.
- **DoUntilContext (s_c):**Timers control under which conditions and how long an action should be active.
- **IntrinsicValue (iv):**To identify the importance or relavance of an action the intrinsic value can be used to direct the action selection.

are the described attributes of the Tuple.

The action selection module uses action groups to select which action to execute. There are two major groups which are used in c4, the attention group and the primary action group. Additional groups for agents are possible but require an extra coordination effort as groups are meant to contain mutual exclusive actions. The attention group is responsible for switching attention towards objects of interest in the environment. The primary action group is responsible for agent locomotion. For each of the action groups, there are fundamentally two lists of contained actions. The "startle" list contains actions which should urgently execute. Thus, as long as elements are in the startle list, they are executed according to their prioritised order within the list. The priority is based on the intrinsic value iv and the trigger context tr_c. If the startle list is empty, elements of the second list—the default list—can take control and execute. Elements in the default are chosen using a probability distribution in contrast to the startle list. The intrinsic value iv in combination with tr_c and s_c is used to create the probable reward with affects the probability of an action being chosen.

c4 allows only a small number of actions to be active at a given time. This is controlled by the number of individual action groups. Actions are evaluated and started when other actions finish their execution or when following conditions are met:

- An action from the default list is currently active but an element is added to the startle list. This triggers a switch in activation and favours the element from the startle list.
- An action receives a boost in attention, at least doubling its evaluated value. This indicates an important environmental shift and actions get re-evaluated.

The learning within the cognitive system is facilitated in three ways. The first mechanism is reinforcing action tuple chains. Actions spread their intrinsic value to actions which lead to the execution of the executed action. This mechanism is monitoring the execution of actions modelling potential action chains—actions which start after a previous action successfully finishes. By spreading iv along those chains priorities within the startle list are reordered. In the default list, this increase in iv leads to higher changes of execution the actions in a chain as well. The second mechanism is rewarding association learning. An example would be when a dog sits down after perceiving the utterance "sit". If the dog receives a reward the association between the utterance "sit" and the action "sit down" is reinforced. This is done using statistical models on the action tuple set and goes hand in hand with their last learning model which is altering the inheritance model of a percept and the linkage to the primitive action a of a tuple. When reinforcing the association an existing action tuple can be altered to reflect a strong association. If the percept "sit down" is an "utterance" and an existing action tuple is

$$A_1 = [\text{"}SIT\_DOWN\text{"}, \text{"}UTTERANCE\text{"}, -, -, 50] \tag{2.5}$$

a new action can be innovated by using the stronger child action from the percept. Thus, creating

$$A2 = [\text{"}SIT\_DOWN\text{"}, \text{"}sit\text{"}, -, -, 50] \tag{2.6}$$

which switches the percept "utterance" against its child "sit" due to the received reward from the system. The three described forms of learning within c4 are all driven by user reinforcement. A similar approach is given in the section on neural networks but is less sophisticated. In games an approach which learns complex control mechanisms is time-consuming and sometimes not desirable. For example, learning wrong associations can impact dramatically on the user experience.

In contrast to most other cognitive architectures, a strong emphasis of c4 is also

put on the last layer, the motor control. The motor control impacts the exhibited behaviour and is similar to approaches in robotics which require fine tuning for each agent. Based on the visual representation, high-level commands such as forward need to be translated into actual motions which give an impression of natural motion. If the high-level commands now combine the rotation of the head while moving forward a complex motion controller is required. This is what the c4 motor system does. A more detailed description of the motor system is beyond the scope of this work but given by Downie [2005].

The c4 architecture combines learning, memory, motion control and action selection into a complex cognitive architecture. Each of the five modules by itself is computationally complex. Initially Isla et al. [2001]; Burke et al. [2001] state the requirement for executing a reasonable amount of complex agents. However, during their experiments—Sheep—Dog and Clicker—they state that the system is able to handle two agents using a scaled down version of the c4 system in addition to a number of flocking agents, controlling the sheep. In a later installation—AlphaWolf [Tomlinson, 2002]—they are able to support six entities within the environment, representing a wolf pack. Nonetheless, this installation requires multiple PCs going beyond the scope of what commercial games can utilise. This result is as expected, taking learning, memory management and a complex action selection scheme into account while following the strict restrictions of a game environment. Other critiques on the system are found in the dissertation of Downie [2005]. Actions within c4 are selected based on their intrinsic value or whenever drastic changes happen. This mechanism increases the risk of oscillation between important mutually exclusive actions leading to dithering. In his dissertation, Downie [2005] states that this can happen quite frequently within the cX systems. Another issue is the strong interdependence of behaviours such as A requires B requires C. A problem which can be addressed by decoupling this relation using a planner or a different approach to agent design. In this context, a planner could be used to find and match preconditions in a more elegant way than strong dependencies, see section 2.2.4. The c4 system nonetheless impressively combines a large number of features into a system able to run within a game environment. The system is additionally not available and maintained making it also due to the usage of Java not applicable for the inclusion in commercial games. The approach is transferable but computationally very expensive as discussed earlier.

2.2.8 A Behavior Language (ABL)

Abl is a reactive planning language written in Java. To create a reasoning agent, the planner constructs an active behaviour tree (ABT) for a given set of agents. The

approach is based on the planning language and agent architecture HAP by Loyall et al. [1991]. Based on HAP, ABL extends its foundation in following points:

- The HAP syntax has been changed for ABL to incorporate object-oriented design features available and supported in JAVA.
- The system allows for a more generalised integration of agent sensory-motor control within the planning language. To support this ABL acts' sub-class the original senses and actions, providing an interface between the environment and the planning language.
- The system supports explicitly multi-agent coordination through *joint-behaviours* which allows synchronisation of memory and behaviours.
- Atomic behaviours lock the current execution order of a behaviour to guarantee its sub-behaviours are executed without interleaving other behaviours.
- ABL contains a blackboard memory system. Working memory elements (WME) are typed data elements which can be accessed from within the planner. Named memory elements can be used by joint behaviours to directly exchange information.
- "Goal Spawning" allows ABL to change the ABT by adding a new independent goal to a different branch of the tree. This can be seen as triggering a different independent behaviour which does not affect the current behaviour.

The motivational goal of creating ABL, as stated by Mateas and Stern [2002], was to go beyond traditional branching narratives from literature and to utilise computational techniques both as support and creative structure. ABL is a reactive planner which allows for plan generation and re-organisation of behaviours to fulfil the current agents' goal. The language uses behaviours as main plan/tree elements. A behaviour can be sequential or parallel . Sequential behaviours are internally pursued in a step-wise process. Parallel behaviours, contrasting sequential behaviours, allow the ABT to branch and pursue multiple parallel goals or actions. If all behaviours within the ABT are sequential, the tree would collapse similar to traditional plans into an execution chain. In addition to the two types of behaviours, behaviours can be joint or *atomic*, as discussed above. Joint behaviours and atomic behaviours require stronger conceptual design. Both synchronize the execution of the ABT to a different degree. Atomic behaviours cannot be interleaved with other behaviours which requires the execution tree to unite into one node at this point. Joint behaviours allow for coordination between a specified set of behaviours, similar to *behaviour coupling* as discussed by Mali [2002]. Mali

```
1  /*
2  * Find idle constructors. (timeout)
3  */
4  sequential behavior detectIdleConstructors() {
5  ProbeWME worker;
6
7  with (success_test {
8  worker = (ProbeWME task==WORKER_CONSTRUCT order==PlayerGuard)}) wait;
9
10 mental_act { worker.setTask(WORKER_IDLE);}
11
12 subgoal WaitFrames(24);
13 }
```

Figure 2-24: A sequential ABL behaviour from Eisbot [Weber et al., 2011]. The behaviour is responsible for detecting idle construction units(Protoss probes) controlled by the player.

however couples behaviours through sensors and focuses on removing internal memory whereas Mateas and Stern uses memory and focuses on interleaving behaviours in a designable way. Both, *atomic* and *joint* behaviours result in distinct patterns of nodes within the tree. Those patterns emerge based on the design of the joint behaviours communication. A heavy usage of joint or atomic behaviours would lead to essentially to deterministic, fixed ABTs.

Nonetheless, they present an important tool for designing interactions and creating designer intended situations or interactions between agents. An ABL behaviour can contain a set of subgoals, WMEs, preconditions, conditions, *acts* and $mental - acts$.

Preconditions are used to evaluate if a behaviour can be used in the currently active context. The ABT in contrast to other reactive planners does not create a plan through back-chaining from the current goal by using preconditions and post-conditions of actions/behaviours. In ABL goals are matched to behaviours by identifier name. Thus, the ABT is created in a forward manner, which is also easier to follow by tracking the logical order of execution.

The behaviour illustrated in figure 2-24 has no precondition and succeeds or fails based on the result of its goal behaviour WaitFrames(). The ABL behaviour describes a sequential behaviour for EISBOT, a STARCRAFT agent designed by Weber et al. [2011]. It is responsible for detecting idle construction units—Protoss probes. ABL offers a way to implement a Listener pattern from OOD reducing the need for constantly checking for a condition. The listener is implemented on line 7 and 8 using the wait function which blocks the remaining behaviour from executing in combination with the success_test . Once the working memory element worker of the type ProbeWME is a constructor and on

guard, the behaviour resumes execution and triggers a mental_act. The mental_act directly accesses the memory and changes the state of the worker assigning it to rid itself of its current task and wait for a new one. The behaviour then waits for 24 frames before succeeding.

To work on designing interactive narratives [Mateas, 2003; Mateas and Stern, 2003], Mateas and Stern developed a design approach using *story beats* which allows to organise the dramatic performance within an environment creating coherent plots. Beats are behaviours arranged around a single dramatic goal. They fall into three categories: handlers, beat-goals and cross-beat behaviours. The handlers are used to track and respond to player actions driving the story plot towards the beat goal. In contrast to other behaviours, handlers are normally persistent . Thus, once they are finished, they reset and continue to pursue their sub-goal. Cross−beat behaviours present side-stories which are useful for enriching the experience and the plot but are relatively short and do not affect the main goal or outcome of the plot. A Beat−goal follows a sequence of interactions between player and agents to communicate or drive a story. A chain of beats creates an experience. As beats can be re-ordered or changed based on the user interaction, different narratives can emerge without the explicit need for an ad hoc creation of fixed branches between the beats.

Mateas and Stern [2003] discuss the design effort of creating *Façade*—an interactive narrative where the player interacts within the social context of a couple on the verge of breaking up their relationship. The issues and hurdles encountered during this process are mainly based on the complexity of designing a large enough set of beats which are consistent. Beats contain hand-authored short sentence-based interactions. Their future work emphasises the importance of an authoring tool to allow even non-programmers to design beats and to support the creation process using ABL. An advantage of using ABL in contrast to fully hand-authored stories is the automatic alignment of beats into stories. Thus, the author has to concentrate only on beats instead of the overarching story and how it can branch. The approach is also applicable for real-time strategy games resulting in a strong artificial agent [Weber et al., 2011, 2010a]. Due to ABL being written in JAVA and the complexity of the approach it is mostly used as an external component which works well but requires a separate process to be run. This makes it harder to include in commercial games.

2.2.9 Behavior-Oriented-Design (BOD)

Bryson and Stein [2001] introduce BOD based on earlier work on behaviour-based robotics architectures [Bryson and McGonigle, 1998; Bryson, 2000b]. In this earlier work, they analyse shortcomings of existing architectures, e.g. PRS [Rao and Georgeff,

1991], and approaches, e.g. SUBSUMPTION [Brooks, 1986], and identify a fundamental issue in intelligent agent research. One of the main driving forces behind Bryson's research is the hypothesis that it is difficult to study theories of cognition when the integration of said theory on top of a given underlying architecture is already difficult. Thus, an underlying architecture which supports easier design of agents should be able to facilitate better research on theories of intelligence. As most behaviour-based agents are using hand-authored behaviours, Bryson and Stein identify the behaviour decomposition as a critical step when developing agents, as well as the absence of a centralised action selection. Brooks [1991] discussed the need for more reactive approaches in robotics systems because deliberative planning approaches initially were too slow for appropriate reactions within an environment and could not handle real-time dynamic environments. Bryson and Stein extend this approach further but argues for a reactive planning approach using hierarchical organisation.

In addition to an architectural component, BOD provides a development methodology to support the design and flatten the initial learning curve. This is done under the previously mentioned premise that to develop intelligent agents the architecture must facilitate the design and aid the developer. The architecture part of BOD uses its parallel-rooted slip-stack hierarchical planner—POSH [Bryson, 2000b]. Nonetheless, the methodology itself is transferable to other reactive BBAI approaches such as ABL, see Section 2.2.8, or BT , see section 2.1.1. To give a more applicable example of BOD, we will start by looking at POSH action selection first and use a POSH example to demonstrate the BOD methodology.

Parallel-rooted Ordered Slip-stack Hierarchical planning

POSH planning is a form of reactive planning [Ghallab et al., 2004], which combines faster response times similar to reactive approaches for BBAI with goal-directed planning. Traditional deliberative planning plans a full action chain from the current state of the world until the goal of the agent is reached without taking in any further information or alteration to the chain once execution starts. This means the planner has to take all environmental conditions into account, an impossible task in a dynamic environment. Whenever the deliberative plan is disturbed, the agent needs to re-plan the whole chain making it a computational costly approach. It is also hard to determine in a deliberative plan when to stop the chain, based on new information. Reactive planning takes a different approach and only plans for the next action towards the global goal—taking the local environment and possibilities into account and not focusing on the global setting. On the one hand, this impacts the optimality of the global plan negatively, resulting often in locally optimal but globally non-optimal behaviour.

On the other hand, the approach allows the system to react better to environmental changes, which are intractable in deliberative planning and allow the planner to operate in a non-deterministic environment. The POSH planner makes use of the reactive planning paradigm and only plans locally, which allows for responsive, yet goal-focused behaviour, required for DEEPER AGENT BEHAVIOUR. Another important feature is the usage of the parallel-rooted hierarchy which allows for the quasi-parallel pursuit of behaviours and a hierarchical structure to aid the design. Bryson [2000b] illustrates that her approach of combining a reactive hierarchy not only outperforms a fully reactive architecture systems Tyrrell [1993] when using BOD, it also shows that a simplification in the control structure can be achieved using her hierarchical approach.

POSH action selection uses five different element types within its reactive plan structure and links the resulting plan through its behaviour primitives (actions and senses) to a behaviour library. A behaviour library is a set of classes based within the POSH library and is responsible for agents' sensory-motor access and memory. Each behaviour class is implemented in the domain specific language, e.g. Java or PYTHON, and should be self-contained set primitive actions and senses which are accessed when POSH is loaded and instantiates for each agent the behaviour library. In JYPOSH—a python implementation of POSH—the planner has access to object methods through named lists for perceptual and action primitives, see listing A-1 on page 230.

- *Primitive (A—S)*: POSH provides two types of primitives: action primitives and sensory primitives. Primitives are leaf nodes inside the plan tree and are implemented within a given behaviour in the behaviour library. Primitives provide interfaces for the plan to their counterparts in the behaviour library. An action primitive is a self-contained piece of source code which controls a part of the agent's expression in the environment. A sensory, or perceptual, primitive extends the functionality of action primitives by including additional feedback to the plan. Sensory primitives are used within the plan goals and preconditions for determining if a plan element should be pursued.

- *Action Pattern (AP)*: Action patterns are used to reduce the computational complexity of search within the plan space and to allow a coordinated fixed sequential execution of a set of elements. An action pattern—$AP = [\alpha_0, \ldots, \alpha_k]$—is an ordered set of action primitives which does not use internal preconditions or additional perceptual information. It provides the simplest plan structure in POSH and allows for the optimised execution of behaviours. An example would be an agent that always shouts and moves its hand upwards when touching a hot object. In this case, there is no need for an additional check in between the two

action primitives if the agent should always behave in that manner. APs execute all child elements before returning.

- *Competence (C)*: Competences form the core part of POSH plans. A competence $C = [c_0, \ldots, c_j]$ is self-contained basic reactive plan (BRP) where $c_b = [\pi, \rho, \alpha, \eta], b \in [0, \ldots, j]$ are Tuples containing π, ρ, α and η: the priority, precondition, child node of C and maximum number of retries. The priority determines which of the child elements to execute, selecting the one with the highest priority first. The precondition is a concatenated set of senses which either release or inhibit the child node α. The child node itself can be another competence or an action or action pattern. To allow for noisy environments a child node can fail a number of times, specified using η, before the competence ignores the child node for remaining time within the current cycle. A competence sequentially executes its hierarchically organised child-nodes where the highest priority node is the competence goal. A competence fails if no child can execute or if an executed child fails.

- *Drive (D)*: A drive—$D = [\pi, \rho, \alpha, A, v]$—allows for the design and pursuit of a specific behaviour as it memorises its execution state using the *slip-stack* [Bryson, 2000b]. The drive collection determines which drive receives attention based on each drive's π, the associated priority of a drive. ρ is the *releaser*, a set of preconditions using senses to determine if the drive should be pursued. α is either an action pattern or a competence and A is the parent link to the drive collection. The last parameter v specifies the frequency which allows POSH to limit the number of executions of the same sub-tree within a given time.

- *Drive Collection (DC)*: The drive collection—DC—is the root node of the plan—$DC = [g, D_0, \ldots, D_i]$. It contains a set of drives $D_a, a \in [0 \ldots i]$ and is responsible for giving attention to the highest priority drive. It also contains a goal which allows the agent to terminate once it is fulfilled. The goal is the highest priority element in the set and is the first element evaluated each cycle. To allow the agent to shift and focus attention, only one drive can be active at any given cycle. Due to the parallel hierarchical structure, drives and their contained sub-trees can be in different states of execution which allows for time-slicing and a quasi-parallel pursuit of multiple behaviours on DC-level.

The plans which are constructed using the described elements are each hand-authored for a specific agent. However, the plans are written in lisp-like syntax which allows them to be learned or generated using computation approaches such as genetic

```
(C get-enemy-flag (seconds 30.0) (goal ((have_enemy_flag 1.0 =)))
   (elements
      (  (CE-moveto-flag (trigger ((selected_target 1.0 =) ))
          moveto-selected-nav)  )
      (  (CE-select-enemy-flag (trigger ((have_enemy_flag 1.0 !=)))
          select_enemy_flag) )
   )
)

(DC life (goal ((game_ended 1.0 =)))
   (drives
      (
      (get-enemy-flag-from-base (trigger ((enemy_flag_reachable 1.0 =)))
                  get-enemy-flag(seconds 0.3)))
      (
      (inch (trigger ((succeed))) AP-inch(seconds 0.3)))
   )
)
```

Figure 2-25: A POSH plan for UNREAL TOURNAMENT agents on a capture the flag map. The plan is controlling a single agent.

programming which work on lisp structures. A reduced plan in source form is given in listing 2-25 and the full plan is available in listing A-2 in the appendix. The same plan as in listing 2-25 is additionally visualised in Advanced Behaviour-Oriented Design Editor (ABODE) and allows for easier visualisation and graphical editing. The sequential behaviour slicing which emerges using the drive collection, by allowing a drive to memorise its state and shift attention to another drive, is biologically motivated and supported by research [Bryson, 2000b].

The combination of behaviour libraries and reactive plans is a strength of the BOD system, it decouples the plan design and the underlying agent environment-dependent implementation. In contrast to other cognitive architectures, memory and learning are not part of POSH or BOD's core system. To reduce the impact on agent performance both systems are intended to be part of the optimised agent code within the behaviour library. This makes BOD the most lightweight approach within our comparison, yet it still allows a BOD agent to contain all parts of a fully cognitive system with agent-specific behaviours.

Iterative Design using BOD

Behaviour-oriented design extends the concepts of object-oriented design (OOD) into the domain of agent development. BOD focuses on rapid-prototyping and iterative development of agents by interacting with behaviour objects. Each behaviour in BOD is treated as a separate self-contained object which can be accessed through methods.

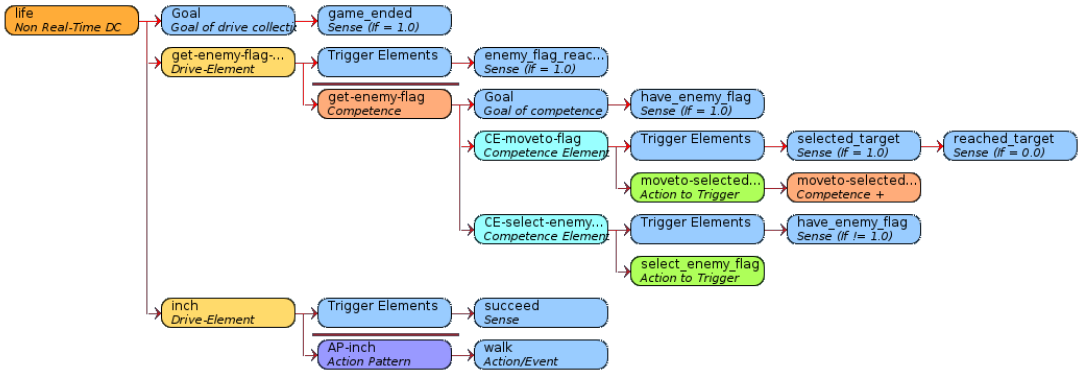


Figure 2-26: A visualisation of the POSH action plan from listing 2-25 using ABODE. The POSH tree starts at the root node—the drive collection—and unfolds for each drive until it reaches the leaf nodes which are behaviour primitives. The plan contains two drives, namely get_enemy_flag and inch.

Similar to OOD, BOD uses inheritance and encapsulation to make the agents more robust by separating independent parts. A crucial step is the behaviour decomposition which is the first step in BOD [Bryson and Stein, 2001].

Initial Decomposition

1. Identify and clearly state the high-level task of the agent.
2. Describe activities which the agent should be able to perform in terms of sequences of actions. Prototype first reactive plans.
3. Derive perceptual and action primitives from those initial plans and sequences.
4. Identify required states and cluster primitives based on shared states into behaviours.
5. Derive goals and drives and order them according to their intended priorities. Prototype drive collections using those drives.
6. Implementation of a first behaviour from the behaviour library.

After the decomposition, we now have an initial POSH plan and a first behaviour. Bryson and Stein [2001] describe the remaining process as non-linear but iterative by repeatedly coding behaviours, coding plans, testing plans and behaviours and revisiting the initial specifications. This process is similar to an iterative Waterfall model [Beck, 1999]. To complete the development process Bryson and Stein [2001] argue for a set of heuristics guiding the development process following the philosophy: "when in doubt, favor simplicity".

BOD Heuristics

- Use the simplest structure first by choosing a primitive over an action pattern and an action pattern over a competence.
- Reuse competence and action patterns as much as possible instead of writing new ones. BOD treats all graph elements as objects allowing the developer to reuse the same node in multiple places by instantiating a clone.
- Decompose actions and senses when only parts are needed and include the decomposed primitives in the tree to replace the original node.
- When an action pattern requires a cyclic call or only sub-parts are always needed change the action pattern to a competence.
- When a competence always executes all child nodes, change it to an action pattern instead to reduce the computational cost.
- If a competence contains two distinct behaviours split it into two.
- Competences generally should only contain between three and seven children to reduce the complexity of the plan.
- If more than three sensory primitives are needed for a releaser or a goal combine them into a single primitive to optimise the plan.
- Primitives should execute fast and not require a large amount of computation. If a larger amount computation is needed, the primitive should be used as a trigger for the behaviour to compute in the background.

The presented BOD heuristics foster simpler structured plans and integrate most of the cognitive tasks into the behaviour classes. This allows the planer to execute and select plan elements faster and the designer to remain in control by not being overloaded with complex, large plan structures.

2.2.10 Generative Agent Design

Research on generative modelling of agents and generative player modelling has seen a recent increase in attention [Holmgard et al., 2014; Ortega et al., 2013; Sandberg and Togelius, 2011; Perez et al., 2011; Grey and Bryson, 2011; Lim et al., 2010]. Generative approaches to agent design [Holmgard et al., 2014] aim to fully automatically build models for more competitive or more appealing agents. Some of those generative approach use utility-based fitness evaluation to achieve better performance. Others are

based on human data. All of those approaches however use techniques from machine learning and AI to learn or evolve functionality not available at its creation time and thereby increase its own or connected agents' capability. The important components in most generative approaches are the fitness function and the learning/adaptation mechanism, dealing with how the agent adapts its capabilities based on better fitting the specific conditions.

One way to create new agents is by utilising human generated input. Using data derived from human interaction with a game—referred to as human PLAY TRACES—can allow the game to act on, or react to input created by the player. A human PLAY TRACE is an ordered set of event which allow us to understand and recount what the player experienced in the game environment. Khatib et al. [2011] use a slightly different approach to acquiring and using human input, in their game "Foldit", they use human players as a crowd-sourced search function. Exploring the search space is costly and by using the crowd-sourced results from player they are able to offload a large part of their computation. However, their search function greatly affects how well their approach is able to explore the solution space.

A general disadvantage of training on human data is that the generated model only learns from the behaviour exhibited in the provided data. Thus, only a small fraction of behaviours from the pool of possible behaviours may have been presented. If the data is sparse the prediction of the human play behaviour cannot be guaranteed to be accurate. Another issue is overfitting the data.

The current approach to counter those issues is to collect as much data as possible and feed it into the learning system, similar to the training mentioned in Section 2.1.3. Stanley and Miikkulainen [2002] use neural networks (NN) to create better agents and enhance games using Neuroevolution. As discussed earlier, this approach allows for the fully automatic evolution of agents which satisfy the criteria of the player judging their performance but the results are difficult to generalise to other games or even different scenarios.

2.2.11 Summarising Agent Design Approaches

In this section the term agent was introduced based on the initial definition by Russell and Norvig [1995] and extended to describe game agents. Additionally, the differentiation between game characters as only the physical representation and the game agent was made to reduce confusions. Game agents are the underlying logical components which control one of many game characters creating the impression of artificial agency. Game agents are able to sense and act in a given game environment but do not have to follow the same restrictions as the player, however for DEEPER AGENT BEHAVIOUR cer-

tain criteria such as situatedness, reactivity, expressiveness, soundness and scalability of the agent need to be considered upon design. After defining the term agent, different architectures from less cognitive to fully cognitive systems were examined focusing on their applicability to games and their approach to developing agents. Based on the amount of conceptual design and complexity fully cognitive architectures are not able to handle the requirements of current games whereas more light-weight approaches lack sophisticated cognitive models. A middle-ground focused on designing complex agents which are able to work under resource pressure in terms of allowed computation time and fixed memory is identified as the best available compromise. Most academic approaches are directed at a special community centred around the origin of the approach which reduces their impact on industrial work. A commonality of approach transitioning to the games industry seems to be a robust architecture that scales to the requirements and has a shallow learning curve; an academic approach matching those criteria is BOD.

In the next section, we will examine existing tools that support the design and development of IVA.

2.3 Game AI Design Tools

The previous section introduced the agent concept and presented different approaches to agent design, from light-weight approaches used in commercial games to more sophisticated, fully cognitive architectures such as SOAR. All of those require to different degrees the usage of programming languages to code agents. In this section, we will focus on tools supporting the development of agents.

Current software for game AI development and DEEPER AGENT BEHAVIOUR does not come with robust tool support. Instead, most of the approaches presented in this chapter are either developed in an existing domain specific languages using Integrated Development Environments (IDEs) for those languages or simply lack any tool support. Even though the available programming tools are designed for experts, requiring an initially steep learning curve and are tailored towards programmers, they still lack enough support. The need for more design support is evident by multiple accounts throughout literature and industry. Thus, if such an approach exists, the design time can be reduced, and the quality of robust agents can be increased. Mateas and Stern [2003] conclude in their work on ABL that the development of their game could have benefited from a dedicated tool to support the development of stories and the ABL behaviour tree. Both authors have worked extensively with the system and can be considered experts. Thus, the issue cannot be related to the initial learning curve of

novice users. Most of the ABL engine is written in JAVA which comes with established decent tools such as ECLIPSE[15], NETBEANS[16], or INTELLIJ[17]. They support a large variety of features to support writing domain specific program code. All three tools feature auto-completion of words which reduces the impact of typos. They support debugging of native applications and the inclusion of test software for unit tests. They also try to support the user by allowing syntax-highlighting, code-folding and class overviews.

Still, there is a need for more support. The latter features are language specific and do not apply to ABL. Mateas and Stern [2003]; Brom et al. [2006] argue the need for individual design and programming support for their specific approaches. Additionally, Laird et al. [2000] present their system and focus on the usage of their specific SOAR environment to be used during development as it offers debugging and introspection capabilities. Their developed agents are not very complex and do not require additional advanced features. An industrial application or the usage of SOAR by novice users would require more support. A similar need is also identified by Orkin [2005] while working on his goal-oriented planning system and the different layers of abstraction for designers.

To support authoring and development of agents for games two different directions exist.

- Provide libraries and tools to reduce the complexity of the design process by taking care of parts of the design work. This approach is ideally suited for novices when they get started with development or when the feature is repeatedly required and needs no further adaptation. For experts, however, fine-tuning parameters and being able to alter most of the agent is essential which reduces their need for tools, which take care of functionality for them.

[15]The Eclipse foundation, `https://eclipse.org/`, provides a framework for java-based IDE creation. The most prominent one is the Eclipse Java-IDE which is widely used in industry and academia. Eclipse provides IDEs for most programming languages and the framework can use used to develop IDEs a new or special languages. The main focus on the Eclipse framework is modularity which allows easy recombination in tegration of new modules. The IDEs in the Eclipse framework are open-source and extentable and are available for all standard desktop operating systems.

[16]Netbeans is another established IDE for java-based software development and similar to Eclipse features support for other languages as well. Netbeans, in contrast to Eclipse is owned by Oracle. Nonetheless, the project is open-source and it is it freely available for developers at `https://netbeans.org/`. Netbeans is maintained by Oracle and is less modular than Eclipse. It offers a stricter interface for plugins, which allows Netbeans to maintain or more directed user experience.

[17]IntelliJ is a propriatory IDE developed by jetbrains, `https://www.jetbrains.com`. The IDE comes in two flarours, ULTIMATE which comes with the latest features and dedictaed support and COMMUNITY which is freely available, does not integrate the latest fixes and only minimal support. The IDE is well designed and comes with good support for all major languages. Due to its closed development the usability is central to IntelliJ which makes it a versatile, robust and well maintained IDE.

- Provide a structured approach to design, similar to the advanced support in IDEs such as ECLIPSE. Those tools allow for code augmentation, debugging and user-controlled information hiding. For agent design, it would involve visualising and augmentation the behaviour and offering different perspectives on the same agent or the interaction between agents and their reasoning process.

The first support type is available in commercial tools and allows the quick inclusion of navigation, automatic path-finding and following behaviours. Autodesk Kynapse [Wallis, 2007]and Presagis AI.IMPLANT [Dybsand, 2003] both fall into the first category. They present tool support for automated path-finding, simple coordination between entities and simple task assignment. Both approaches are able to handle large amounts of agents. AI.IMPLANT is used in large simulation environments for crowd modelling. Both approaches are also not free making it hard to use in smaller teams or in teaching environments as discussed by Gemrot et al. [2009]. Additionally, both approaches focus on spatial reasoning which is not that important in games such as Façade or when special requirements need to be met such as a low-performance system or exotic hardware which may not be supported by those tools.

The second type of tools is aiming to support the process of writing AI parts such as SKILL STUDIO, DI-LIB and BRAINIAC DESIGNER. All three tools are aimed at BT implementations of AI systems, see Section 2.1.1.

2.3.1 Pogamut

Pogamut is a framework based on plug-ins for either ECLIPSE or NETBEANS, see figure 2-27. It is built around the UNREAL TOURNAMENT game environment which, due to its open interfaces and extendability, offers a robust basis for experimentation and development of spatially-situated, real-time agents. To allow agents within UNREAL TOURNAMENT to be controlled by external means, Kadlec et al. [2007] use the GAMEBOTS interface module [Adobbati et al., 2001]. Pogamut is, at the time of writing, in its third version [Gemrot et al., 2009] and includes different modules for action selection, such as the previously mentioned SOAR, ACT-R and POSH, and modules for developing experimental settings.

Pogamut's main aim is to provide a freely available environment for rapid prototyping of AI, yet, include advanced features such as behaviour debugging, logging and test-driven development which are normally only found in commercial tools. The target audience is novice programmers interested in the development of virtual agents as well as researchers from other backgrounds than computer science. Pogamut's intended workflow is described as a Waterfall model.

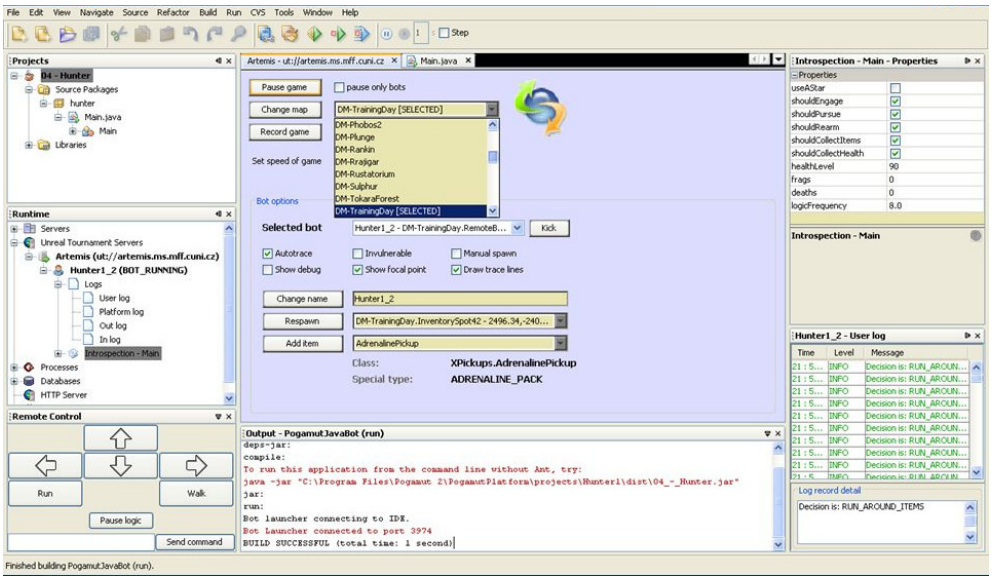


Figure 2-27: The Pogamut framework by Gemrot et al. [2009] for UNREAL TOURNAMENT. The framework is realised through the netbeans plug-in which adds additional features such as the spatial agent control in the lower left of the agent introspection in the upper right part.

Pogamut Work-flow:

- Develop a description of the agent model.
- Implement the agent model within Pogamut by first selecting a decision making approach: ACT-R, POSH, SOAR, or hand-coded in Java or Python.
- Debugging and repairing the implementation is done after a model has been implemented and the test environment can be started. The debugger provides support for inspecting the agent and rerunning the agent which allows the user to identify development and implementation issues.
- Once a reasonable agent has been implemented the parameter can be tuned individually during execution of the agent. Pogamut provides code annotation which allows advanced run-time inspection from within the editor.
- The last phase of the work-flow is setting up experiments for testing the agent. Pogamut offers functionality to re-run and log experiments many times which is useful for large scale simulations. Additionally, the system includes a testing module to support test-driven development.

This work-flow however does not go into the detail of specific agent development which is crucial for novice users. A big disadvantage for novice users is the complex

set-up of Pogamut which requires the installation of different software components and their system privileges. Gemrot et al. [2009] discuss more details of their system but also state that the system is currently not robust and contains software bugs. A stable and robust environment is critical for the usage by novice developers as it can impact their personal motivation and engagement with agent development.

2.3.2 ABODE and POSH

ABODE is an editor and visualisation tool for BOD, see figure 2-28. It features a visual design approach to the underlying lisp-like plan language POSH. The editor is environment agnostic and does not integrate with any other tool which allows a more flexible approach to its usage. Due to its pure GUI-based programming, mistakes are either on a logical plan level or when called the POSH primitives, e.g. actions and senses.

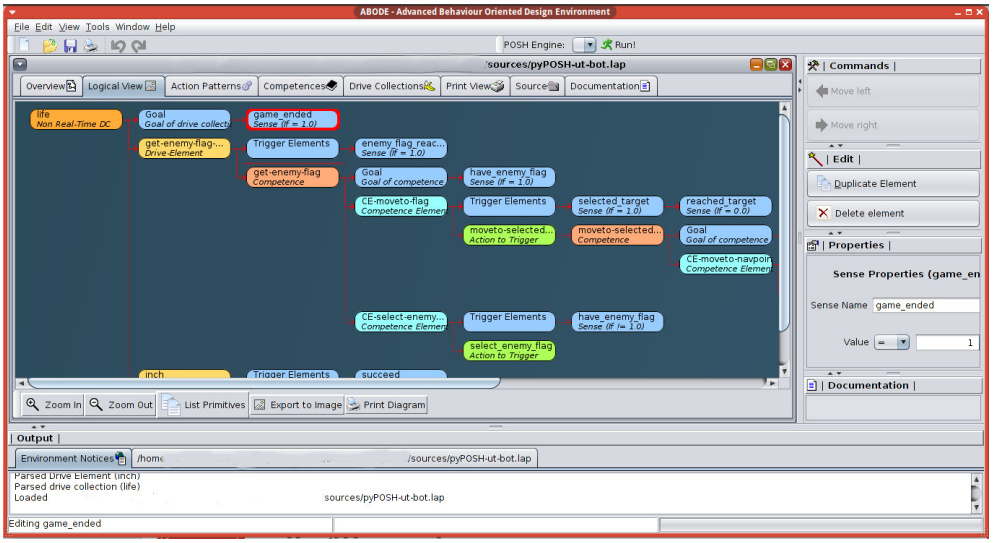


Figure 2-28: ABODE is a behaviour design editor for BOD and allows graphical modification of lisp-like POSH plans. The editor offers different perspectives on the same plan structure allowing the user to only view a limited set of nodes to minimise his or her cognitive load which is a major point of Bryson and Stein [2001] for the work on Behaviour-Oriented Design.

ABODE was designed to support the developmental approach of Behaviour Oriented Design from the initial drafting of an agent through testing until tuning of specific behaviours. It features six essential perspectives on an agent plan and a panel on the right side for modifying a particular node and its position within the plan.

- **Overview:** This view presents a high-level view with reduced visibility of the plan. It illustrates the hierarchical organisation of drives in the drive collection

and the connected competences or action patterns to each drive.

- **Logical View:** The logical view presents the whole plan in a tree structure. This view can be quite demanding once a plan reaches a reasonable size and cannot be visually represented on a single screen. The benefit of this view is to understand where competences are used and if parts of the tree resemble each other and can be combined.
- **Competences:** The view presents the competences separated from the rest of the graph. By only presenting a single sub-tree the user can on individual behaviour development and a narrower focus on its specific sub-tree elements for a given context.
- **Action Patterns:** The view shows all action patterns in a similar way to the competences which is useful for identifying duplicated patterns or patterns which are highly similar and can be merged.
- **Source:** The source view presents a non-editable version of the underlying lisp-like plan. It does not feature syntax-highlighting or other text editor features.
- **Documentation:** The documentation is useful for describing the intended behaviour of the plan as well as the setting and the reasoning behind different decisions.

The general work-flow when using ABODE is to iterate over the drive collection whenever a new feature should be added and include new actions and senses. Then, build up more complex structures such as action pattern and competences. Once a plan file is saved, the required underlying behaviour library, see Section 2.2.9, needs to be checked regarding the used actions and senses. If the plan uses actions and senses which are not available in the library, they need to be implemented before the agent is able to use the designed plan. After finishing designing and saving a plan, it can be copied and referred to by the POSH planner. The editor is stable and is freely available but does not include advanced features such as a debugger. A similar editor for BT would be the Brainiac Designer, see figure 2-30 on page 114.

2.3.3 Visual BT Editors

There exists a large number of different editors and environments for building agents, however, only a small subset of those is recognised by a larger community of users. The ones we will have a closer look at next represent the most promising ones found and accessible at the time of writing. Tools which are only mentioned in research papers

and are not available for evaluation have been not included because they do not offer the possibility of a dissemination by others and are less likely to affect advancements in the field.

BEHAVE is actively maintained and developed by Emil Johansen[18] for building graphical BTs for the Unity2 game engine and has established itself as one of the main commercial BT implementations for Unity. BEHAVE is a stable, minimalist, visual design tool for agent development in Unity and is frequently presented at industrial conferences and workshops such as AIGameDev-Conf and the Game Developers Conference (GDC). The BT editor allows the development of BehaviourTrees of the second generation based on the specification by Champandard [2007a].

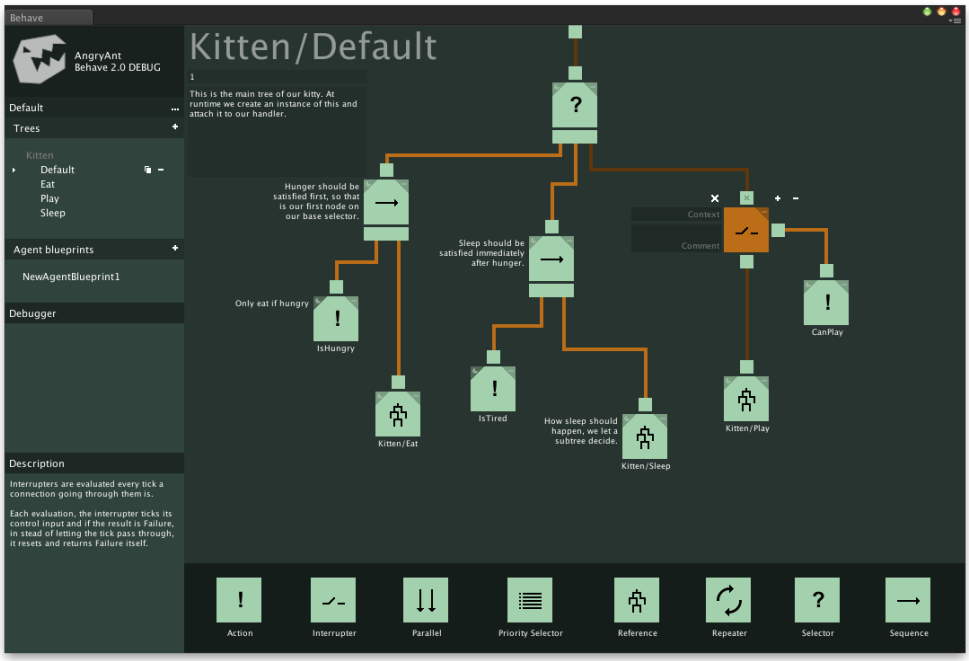


Figure 2-29: The BEHAVE Unity plug-in for developing BTs in the Unity game engine. BEHAVE is fully integrated into the engine and focuses on a simple clean behaviour representation.

BEHAVE is a commercial product and is focused on smaller industrial development teams using the Unity game engine. Thus, it is directed towards a specific target group compared to the later tools. It provides a set of different node types which can be combined into a tree structure to control a game agent linked to the tree through the UI. The strength of BEHAVE is its deep integration into Unity, which allows drag-and-

[18]Emil Johansen is a key figure in the Unity developer community and responsible for the success of Unity's standing as a productive and easy to access development tool which is also because of BEHAVE as one of the first BT IDEs for Unity.

drop support of nodes within the tree. Due to the available example projects, tutorials and the online community, Behave can be used by novice programmers after a short learning period. Once a tree is created, the leaf nodes need to be implemented in the underlying domain language to control the agent. This step of integrating the lower level implementation is supported using either Microsoft's VisualStudio or Xamarin's MonoDevelop in combination with Unity. The linkage between Unity and the software IDE's is a major strength of Unity as it allows the usage of Debuggers from within Unity through both IDE's.

This approach of first designing a tree and then implementing the underlying primitives is similar to ABODE, where the designed plan needs to have a primitive set of actions and senses available but is developed independently. Similar to its underlying approach—BT, Behave does not come with an explicit development methodology or workflow which requires the integration into any other process. An additional design methodology would be able to guide the design. Otherwise, the traditional approach in games is mostly focused on software development approaches familiar to the developer, leaving the designer as someone who is not actively integrated into the software production. With Unity and Behave, the designer will layout the intended behaviour and might be able to receive a test repository. However, he or she will most of the times not be able to contribute to the actual working code base. This and similar arguments have frequently been observed at game developer conference talks given at AIGameDev. As Behave is integrated within Unity, any change in agent behaviours and the inclusion of errors affects the overall stability of the whole game which makes modifications and experiments with different approaches affect the entire team's workflow.

Skill Studio is another design tool and a free plug-in similar to Behave when it comes to the intended target audience of Unity developers. In contrast to Behave, it is open source and available under the Microsoft Public License. The availability of the sources allows a developer to modify and enhance the framework even for commercial contexts. It provides an integration into Microsoft's VisualStudio for programming and building Bts, but similar to Pogamut it is not stable and features outdated instructions which make it less appealing to novice users.

At the time of writing, Skill Studio was not adjusted to work properly with the current Unity version and requires some internal changes; it also is only maintained by a single person and updated infrequently.

Brainiac Designer, see figure 2-30, is another Bt design tool, closer to ABODE than Behave, to the extent that it focuses mainly on the graphical design of a Bt instead of an integration into an existing environment.

Brainiac Designer is stable and easy to use if the user is familiar with BT.

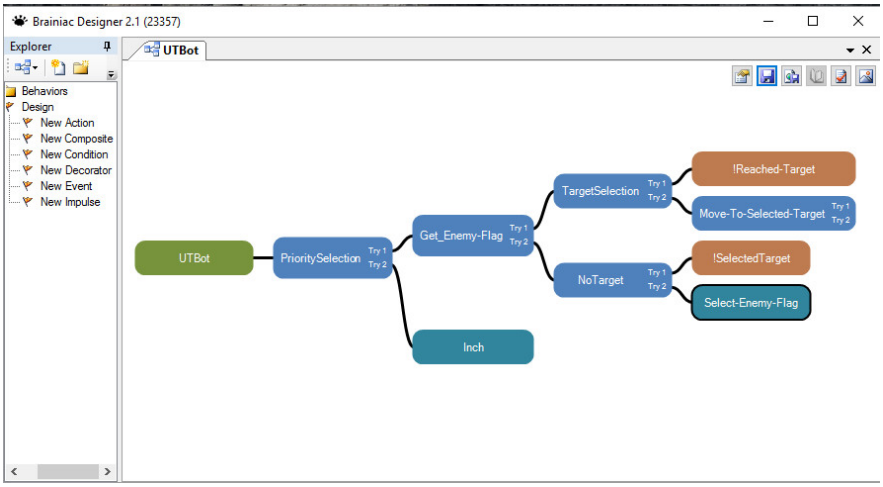


Figure 2-30: The Brainiac visual BT editor is a free xml based tool for designing and exporting agent behaviour. It is freely available at: `https://brainiac.codeplex.com`

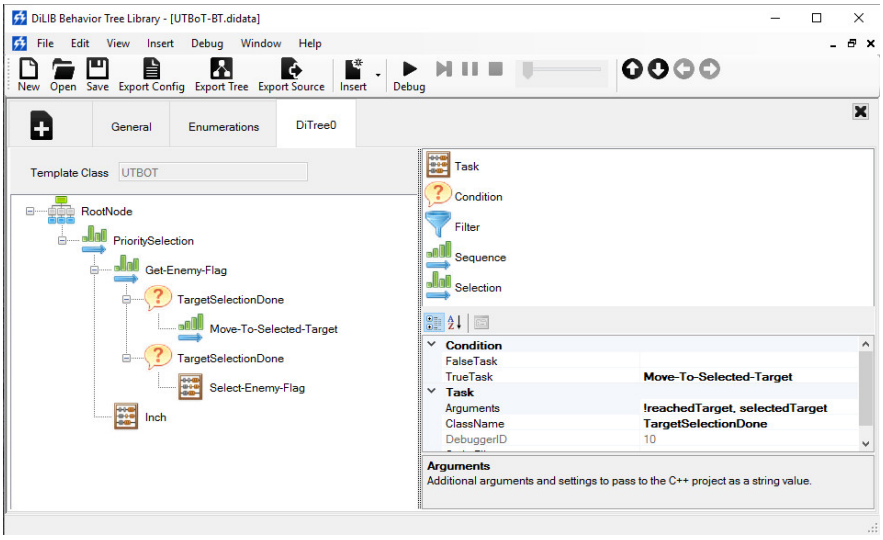


Figure 2-31: The DiLib framework allows the development of BT-based agents and integrates a given C++ behaviour library into its agent framework that can be used within environments which allow the inclusion of external libraries. It is freely available at: `http://dilib.dimutu.com/`

The UI is clean and allows modification of the tree. The strength of the designer is that it exports the BT as an XML document which makes it environment agnostic. Due to the provides source code for it, a different exporter can be written to support a specific BT implementation more closely. Calling and linking underlying actions and perceptual primitives have to be provided within the target environment. DI-LIB which is illustrated in figure 2-31 integrates deeper into a target system by using C++

agent behaviour libraries for actions and senses and providing linking tasks (actions) to specific methods within the library. DiLib also provides a blackboard system and a Bt debugger which allows real-time visualisation of the tree and the state of tree nodes. The framework is well maintained but requires an initial understanding of Bt and is more directed towards designers with a minimal knowledge of programming. It potentially integrates well into an existing workflow and due to the separation of tree logic and underlying API, it should be more accessible to designers and programmers, sharing the work.

Most game-based editors are available for the two dominant development operating systems, Microsoft's Windows and Apple's OSx. Pogamut and Abode also work on any system supporting Java making them marginally more versatile. The tools support laying out behaviour in a graphical way which makes the design shift from pure code-based programming to visual-programming. Visual representation of source code in tools and the needed support as discussed by Storey et al. [1999] highlights elements which are also of importance to agent design tools and show that hierarchical representation which is given using a Bt editor or ABODE can aid program comprehension and thereby aid the design of better agents.

In this section we examined different agent design tools ranging from IDEs such as Pogamut to pure visual editors such as ABODE. The presented tools represent a subset of existing tools but present the most referred to tools by game AI developers. In the next section the chapter will now draw in the findings of all preceding sections to create a comprehensive summary of the state of the art in game agent design from both an industrial and academic perspective.

2.4 Summarising the State of the Art

Looking back at the introduced concepts and techniques, we can now compare the different approaches to agent design. The underlying low-level techniques such as utility modelling or potential fields are by themselves not able to produce complex, DEEPER AGENT BEHAVIOUR for current games. Utility modelling is perfect for modelling numerical, measurable attributes which require low abstraction from the actual environment, such as income or health points and produce high-level priorities for favouring strategies. Nonetheless, utility models need a lot of tuning because they not only contain description of the highest utility of a task but also parameters for balancing them to players' skills. Potential fields, on the other hand, require less manual tuning but similar to other spatial approach are less useful for non-spatial interactions. Those techniques represent base level techniques which can be integrated well due to their

closed systems design. FSMs which are also low-level but allow for better abstraction and additional modelling, they are still the "go-to" approach for simple agent design. Agent and behaviour-based systems widely use FSMs such as the SUBSUMPTION architecture by Brooks [1986]. Even for more complex systems such as the one described by Bojic et al. [2011], FSMs are easier to understand, initially easy to model and well documented, thus, are therefore used frequently by professionals. Due to the discussed problem with scaling FSM-based systems, BTs and other hierarchical approaches such as POSH can be employed in larger scale systems. BT and POSH are only supporting techniques similar to FSMs when it comes to designing systems. However, POSH in combination with BOD provides a design methodology and design support in addition to the software framework.

Mid-level approaches which go beyond development techniques include BOD, ABL, Goal-Driven Autonomy (GDA) and GOAP, they allow for complex agent design and provide a certain degree of guidance for system design. GOAP provides a reactive planner and has shown its application to commercial games. It provides the lightest support for modelling and requires a solid understanding of programming to develop interactive behaviours based on the underlying planning language. The approach also lacks design and novice support, which is a critique affecting more approaches as well. The statements from Section 1.2.1 about shared tasks in programming teams indicates a need for support as well. In most of the game development environments, designers only have a limited interface to the system which hinders creative expression drastically [Anguelov, 2014]. ABL, in contrast to that, is intended for programming-literate designers who want to express themselves through the usage of AI [Mateas, 2003]. The approach is worth striving for but too demanding to be used in current game development. The current ABL system itself is also rather hard to use in commercial settings or systems with a low computational power footprint. Those disparities are due to the requirements of the Java virtual machine, the time for re-planning and the development and set-up procedure. ABL and GOAP also present planning approaches without any visual design support, creating a barrier for non-programmers or when highly complex structures are developed. There is a high similarity between GOAP/FEAR [Champandard, 2007b] and ABL when it comes to deliberating a goal which makes them comparable regarding potential performance and illustrates that if ABL would exist in an easier to integrate module it would be more applicable to commercial games.

Out of the above, only ABL, BOD and Pogamut include explicit design approaches guiding a user to create agents in a specific way. However, the Pogamut design methodology does not go beyond a mere list of high-level points and the documentation by Gemrot et al. [2009] does not guide novice developers appropriately, as essential steps

of the initial design and how to incrementally address design issues are left open. ABL, developed with narrative generation in mind, introduces the BEAT idiom which approaches the structure of interaction between the agents and the player in a drama-focused way. The BEAT allows the design of narratives and interactions between agents and the player in a way which is familiar to writers. The implementation of sequential and parallel behaviour is not guided and is harder to design once the system reaches a certain complexity as the behaviour tree underlying ABL can be changed at run-time by any behaviour. BOD puts a clear focus on the entire design process and presents guidelines for the segmentation of tasks into atomic elements, used within the architecture, to build an agent. It is also the most light-weight approach, similar to BT. All other approaches present implicit design rules which are not transparent and guide the design whereas BOD tries to minimise and off-load as much complexity as possible into more manageable autonomous behaviours.

High-level approaches include SOAR, ACT-R, ICARUS and the cX systems of MIT. Those systems do not scale to large amounts of complex agents but focus on either highly complex single agent approaches, or smaller amounts of agents which require a high-powered system. Production rules are the basis for the first three systems which make the reasoning process understandable, similar to planners where the plan steps can be made accessible in the form of human-readable source code. Due to the integration of learning and the modelling of cognitive processes when acquiring information from the environment and memory, the fully cognitive architectures are hard to integrate into experiences that can be designed. The systems are complex and require high-specialised background knowledge, which is also one of the reasons they are cultivated in static communities. Additionally, writing large amounts of production rules which form a coherent thought process and anticipate the agents' reasoning process, once the developer is supplied with a user, might not be in the interest of game designers when aiming for a variety of interactions and entities.

In their work on SOAR agents, Laird et al. [2000] make assumptions about the quality of commercial games such as, "... games such as Adventure, Zork, BladeRunner, and the Monkey Island series. One weakness of these games is that the behaviour of non-player AI characters is scripted; so that the interactions with them are stilted and not compelling." However, those assumptions are not supported by evidence in the presented work and do not reflect the commercial and social status of those mentioned games. As described by Mateas and Stern [2003], games communicate a story which can be driven by agents. However, the story can also be compelling by itself and well written scripted characters are a means of expressing that. As long as the difference between a well-scripted agent and a fully cognitive human-like reasoning agent are not

detectable there is no benefit in using the second one [Orkin, 2005].

Laird et al. [2000] makes additional assumptions about the complexity of behaviours for games such as UNREAL TOURNAMENT or Quake. They identify the lack of social interaction as the limit for creating compelling bots. However, for fast reaction based games a lot of work is needed to create sufficiently complex underlying operators first before being able to develop higher level motives. Mateas and Stern [2003] discuss the development time for their game which approaches the complexity of commercial projects and they point out the significant amount of effort it takes to pass the threshold of reaching commercial-like levels. Generative approaches using evolutionary methods are hardly used in finished games, either because they require a considerable amount of prior specialised knowledge or because they are too risky concerning a predictable outcome. Vanilla approaches given in introductory texts such as the ones by Sweetser [2004a]; LaMothe [2000] are not sufficient to achieve reasonable performance in commercial environments. During production, some teams use evolutionary systems for parameter tuning as it provides a robust closed-box approach [Brandy, 2010].

Looking at the stages of game development, an approach using rapid iterations of prototypes and a flexible structure is mostly used and favoured. Thus, agile methods such as SCRUM, see Section 1.2.1, are game developers favoured approach.

There exist a couple of open questions in the design of games which require support and are hardly addressed in given methods. Those questions are related to the designer inclusion into the development process O'Donnell [2009, 2012], the availability and usage of better development tools facilitating design Mateas and Stern [2003]; Anguelov [2014] and a general absence of support for novice users. Visual tool support using editors is a first starting point but will not be pursued in this thesis because we must first focus on the underlying structure to support a robust platform before creating visual tools which enhance the process.

In this chapter, we surveyed the existing literature and techniques on game AI design and implementation. We started by looking at low-level techniques such as FSMs, that are well-used by designers and programmers, to evolutionary approaches such as neural networks, which are only usable by programmers and have a steep learning curve. We then discussed a new definition for game agents and how game developers are aiming for DEEPER AGENT BEHAVIOUR to keep the user immersion as high as possible. Based on the new term, we analysed high-level approaches, integrating the discussed low-level ones into an agent framework. Two points of the presented approaches became visible. The first, the more sophisticated the underlying approach is, the more CPU is needed to control sophisticated agents. The second, most of the presented approaches only provide technical frameworks with no explicit support or method to designing agents.

The first point leads to the fact that the game industry is focusing on more light-weight techniques. The second point leads to a reduced impact of academic approaches on the industry and unnecessarily steep learning curves when approaching agent design because the implicit design rules of a given approach have to be acquired through trial and error. This curve is dampened through the usage of tools that were presented in the last section of this chapter.

This survey identified essential elements needed in game development for supporting agents development. Based on strict resource limitations light-weight approaches have been favoured by the industry. More advanced techniques need to be well documented and contain showcases before they are employed by game developers. This suggests that the learning curve of using a new approach impacts the decision of selecting appropriate approaches. Because games are developed in multi-disciplinary teams, the development model needs to support the different tasks instead of simply forcing all team member to rely on other team members. As discussed in this chapter, most of the frameworks are programmer centred, to support designers and to guide the development, we identified that a new process model is needed which integrates both designers and programmers. Additionally, due to the complex nature of game development, more complex encapsulated solutions such as potential fields or evolutionary approaches are only used if the risk of integrating the approach is low or predictable, for most academic approaches this risk cannot be predicted because show-cases or demonstrators rarely exist. In Chapter 5 those elements are realised in a new approach to designing game AI—Agile Behaviour Design. The novel methodology focuses on shared tasks and strong guidance for developing sophisticated agents and is demonstrated in two showcases. To support the robust development, advanced features such as *behaviour inspection* have been integrated into a new supportive framework, posh-sharp which creates an industry compliant light-weight framework.

In the next chapter, we will investigate three frameworks through a set of case studies and informal expert interviews of authors. Thereby, we identify the implicit design approaches of each framework and derive a methodology which can be used with other IVA architectures to analyse their approach to design.

Chapter 3

Requirements for Agent Tools

In the previous chapter, we surveyed the state of the art in game AI techniques and approaches for agent design und how it relates to understanding and developing new methods for robust agent design. The survey spanns from low-level approaches for decision making and spatial approaches from physics and robotics to high-level frameworks from Cognitive Science such as ACT-R. This comprehensive view of game AI is needed to understand the complexity of developing approaches for game agents. As a result of the survey, we identified essential elements for developing game AI, limitations of the existing approaches and reasons for the spread of architectures—why only some are employed outside of their original development community. In this chapter we examine three prominent Interactive Virtual Agent (IVA) frameworks to further extend our knowledge on the requirements for IVA development and present case studies which identify underlying similarities and problems of agent design which are essential to the development of new tools and approaches for IVAs. This knowledge will be utilised in Chapter 5 where we propose a new method and framework for agent design that integrates the findings of this chapter.

3.1 Contribution

This chapter is based on work that was undertaken in cooperation with the Expressive Intelligence Group at the University of California, Santa Cruz [Grow et al., 2014]. The central contribution of this work is an analysis of the development and design work-flow in three distinct architectures for agents. To achieve this, we conducted informal expert interviews to understand essential approaches for each individual platform and, as a result, compiled overarching strategies for creating intelligent virtual agents for game narratives. The two most notable architectures Behaviour-Oriented Design (Bod) and

ABL, see Section 2.2.9 and Section 2.2.8 respectively, have been discussed before. The third architecture is **F**earNot! **A**ffec**t**i**v**e **M**ind **A**rchitecture (FATIMA) described in detail by Dias et al. [2014]. My contribution to original paper was 30%. However, this chapter is significantly expanding the content and discussion on IVA design.

3.2 Problem Description

IVAs are embodied humanoid characters that are designed to respond richly to user interaction. They combine work in AI, human-computer-interaction and graphics, as well as interdisciplinary knowledge from fields such as psychology and performance arts into a system which interacts with human players. The contained behaviour within each IVA aims to suffice the DEEPER AGENT BEHAVIOUR criteria. To understand the construction of agents, let us define the term *authoring* as a process which encompasses any asset creation and modification necessary to produce the desired functionality of IVAs, such as animation, audio, written dialogue, behaviours, goals and other more specialised decision-making components belonging to a DECISION-MAKING SYSTEM (DMS).

IVAs share the same authoring burdens as animated characters in movies or fully scripted cut-scenes in digital environments, such as animation and dialogue assets. Additionally, IVAs need a DMS to handle the interaction with other agents or a user/-player. This interaction adds another dimension of complexity to authoring. The combinatoric interaction possibilities, including large internal (to the agent) and external state spaces, make it difficult for an author to reason about and modify a DMS without external help.

Authoring tools are often proposed as a means to help the author manage the complexity of the authoring process. Chapter 2.2 offers a more detailed view on different approaches to agent modelling and introduces several tools and platforms which can be useful for modelling IVAs. For authoring tools to be of any practical use, they must be flexible enough to allow for specific domain knowledge to be integrated and to customise a target system towards the project's needs, including the authoring challenges of a specific DMS. The lack of any cross-system tools for IVAs illustrates this problem. The editors and tools discussed in the previous chapter focus on some aspect of the task at hand, still, they are unable to address the full development cycle of creating IVAs. The closest existing tool is Behave—an integrated graphical BEHAVIORTREE tool for Unity— but firstly, it is bound to a particular game engine and secondly is not available for free. Additionally, Behave does not provide a structured design approach

and is more focused towards programmers than authors. Adding to that, it is also not separating DMS and underlying implementation well which makes the design heavily dependent on correctly working behaviours, not to break the environment. Pogamut as a second authoring tool is also tied to a fixed game engine and more specifically to and is not stable and well-documented enough without investing further work.

To understand the authoring process 11 DMS authors across five institutions and nine different projects in the field of IVAs were surveyed. Out of those, three teams were chosen for iterative interviews, where a similar pattern of difficult design decisions was discovered— the pattern is coined the System-Specific Step (SSS).

The SSS describes the DMS-dependent combination of architectural affordance and authorship in which the authors express their high-level vision for the character into a decision policy expressed in a particular architecture. After returning to the three teams with the interpretation of their system's SSS, they confirmed the requirements the SSS places on any authoring tool approach in combination with authoring tool proposals based on the challenges discovered in the System-Specific Step.

The SSS requirements analysis methodology is being proposed as a means by which programmer-authors may better understand their particular system's authoring burden and potential features of authoring tools which would alleviate this burden. Three case studies of agent architectures have been conducted, comprising of different design philosophies, teams and levels of complexity as rigorous example test cases of the new methodology. The driving force behind them, the support of the creation of authoring tools in similar architectures and enable the authoring of more robust IVAs.

3.2.1 Related Work

For this work, the definition of author is narrowed down to programmer-author; the designer with an authorial vision who has enough technical knowledge to build or use sophisticated or programmer-oriented authoring tools. Including a wider audience of non-programmers would have been desirable to get more general results. Even though, the studies only used programmer-author it was still possible to take some of the wider audiences issues with authoring tools into account by incorporating the findings of Spierling and Szilas [2009]. The process of defining the SSS and tools supporting it involved iterative discussions with the intended authors in order to "make tools that better match the concepts and practices of *our* media designers and content creators" [Spierling and Szilas, 2009].

Even though the content matter of the tools was different, the iterative case studies regarding design support tools for digital games by Nelson and Mateas [2009] is a compelling structure and is used in the presented AI architecture authoring tool analysis.

Based on it, a methodology was built and tested with the available subjects using tight collaboration, which proved key to the success of the carried out analysis.

One of the clearest cases of authoring tool effectiveness was demonstrated by Narratoria, a tool suite that enables non-technical experts familiar with digital media to create interactive narratives [Van Velsen, 2008]. Narratoria is comprised primarily of three separate tools: story graph, script and timeline editors all linked with collective underlying data structures. While the interaction with the created agents is minimal, the addition of the Narratoria tool suite to the agent authoring process reduced the time spent authoring between two similar projects. Narratoria's *divide-and-conquer* approach to authoring tool design, creating each sub-tool with familiar vocabulary and tropes of its specific genre to better support specialised authors, informed the conceptualisation of the SSS.

Another related project is AIPaint [Becroft et al., 2011], a BehaviorTree (BT) authoring tool for creating spatial navigation rules. However, AIPaint only applies to spatial reasoning similar to the low-level techniques presented in Section 2.1.2, rather than social reasoning illustrated in Prom Week by McCoy et al. [2013]. In contrast to the other BT editors that were discussed in the last chapter, AIPaint comes with its own automated approach to designing agents which is based on drawing connections on a screen and then allow the system to infer meaning from the connected elements. Due to this narrow design focus on spatial reasoning for a specific game, it is hard to generalise and has not been included it in the initial analysis in Chapter 2.3.

Learning by demonstration is also an intriguing authoring approach that has been gaining popularity in recent years [Argall et al., 2009], similar to the generative approaches discussed in the Chapter 2.2.10. However, attempting to encode complex human-like decision-making for embodied characters is far beyond the current capabilities of existing systems and requires a considerable amount of training and verification of the intended behaviour. Finally, as AI research progresses, commercial AI systems in games also evolve using techniques from research to empower their systems. Due to the AI challenges in games in recent years BTs, see Section 2.1.1, have become one of the most dominant industrial approaches to structure and control intelligent agents in games; two of the conducted case studies employ techniques similar to BT in combination with reactive planning.

3.3 The System-Specific Step

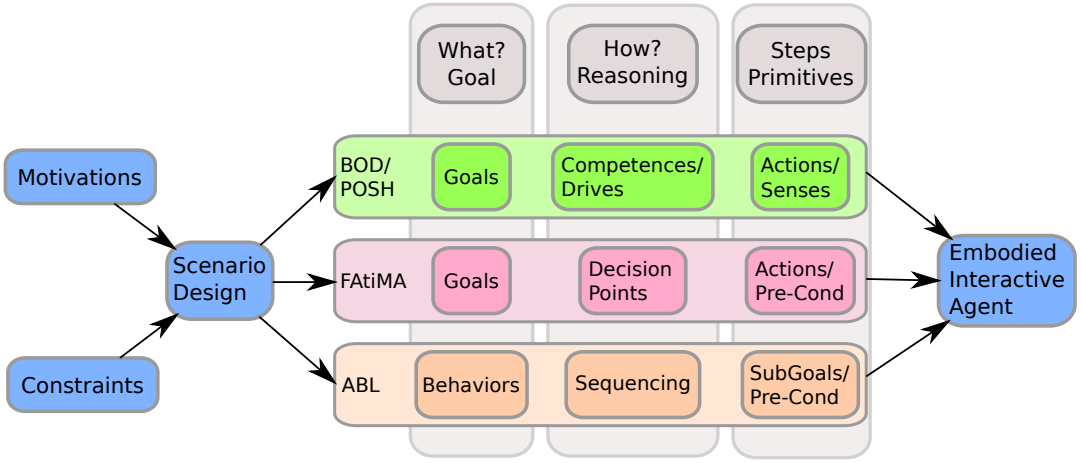


Figure 3-1: Representing the System-Specific Step for creating an IVA with BOD,FatiMA and ABL. Each SSS specifies which goals the agent needs to achive, the approach of how the specific system is able to pursue those goals and the elements the agent executes to achive its goals. The three areas which form the SSS can be found in each of the three systems discussed in this chapter. For POSH-SHARP they are illustrated in figure 5-3 on page 177.

For working on IVAs, it has been identified that each system comes with its own, most of the times implicit, design philosophy, coding style and structures as a primary authoring challenge. While all share the concept of an authoring burden, how this burden manifests in each system can be entirely unique. In order to begin easing the authoring burden for each system, the peculiarities of the authoring burden in specific instances have to be identified first.

The SYSTEM-SPECIFIC STEP is the term for the parts of the authoring process where the general discussion and design reach their limits and, as the name suggests, system-specific design constructs are used instead. Any design or production for a particular system regarding that system's unique architecture, design constructs, code objects, or design philosophy is part of the SSS of that system. The SSS is where all the gritty intermediate- and expert-level authoring (in design and code) takes place to make the interactive agents and experiences a reality.

Examples of what authors may do as part of their SSS include:

- imagining how an agent will traverse a behaviour representation so the author can craft interesting decision points

- construct hierarchical goals so that an agent can plan its way from the beginning to end of a scenario
- figuring out how the agent can express frustration if its body is busy doing other actions

The more explicitly and concretely an SSS for a system can be defined, the clearer the problems that an authoring tool may attempt to alleviate.

3.4 Interview Methodology

To explore the authoring process of embodied interactive characters, a series of informal interviews with five institutions across the globe were conducted to help understand how reasonably isolated groups approached their personal authoring challenges. In addition to the six local ABL authors and the team involved in this project, members of GAIPS Paiva [2013], CADIA University [2013b] and CTATUniversity [2013a] were surveyed, to explore different approaches and purposes for authoring. Those purposes include authoring-by-demonstration for educational purposes, as well as creating tools for various levels of author expertise. These programmer-authors also shared anecdotes of successes and failures of particular authoring tools, approaches to authoring tools and agents and techniques for visualisation. The resulting findings backed up claims made by the field in general, namely that "authoring is challenging and in need of help via tools". These support the proposal of the idea of the SYSTEM-SPECIFIC STEP (to describe the different, yet similar, phases of the authoring process and the requirements they present for any authoring solution) and lead to utilising the SSS for the design of authoring tools using in-depth case studies.

After distilling the information from those interviews, the teams willing and able to conduct follow-up case studies were re-visited regarding their SSS. Each of the three programmer-designer teams that volunteered for the case studies was given the same simple scenario, described below. For that purpose, they were asked to transform the scenario into descriptive pseudo-code for their system, one step removed from actually programming the scenario. The procedure the teams followed to create the pseudo-code was simultaneously translated into a rigorous process map[1]. Details of each step (and possible sub-steps) in the process were recorded, such as the duration of each step, the involvement of other people and potential authoring bottlenecks.

The resulting process maps of each case study looked drastically different, and it proved hard to retrieve time estimates from the interviewed authors. However, it was

[1]Process mapping involves creating a visual representation of a procedure similar to a flow chart, making explicit "the means by which work gets done" Madison [2005].

possible to extract sufficient information to construct a SSS for each system and propose authoring tools to alleviate concrete issues discovered in the process map.

3.5 The Scenario

The scenario that was chosen is a simplified version of the "Lost Interpreter" scenario recently completed and demonstrated within the IMMERSE project which uses ABL as its DMS [Shapiro et al., 2013]. The scenario involves the player as an armed soldier in an occupied territory searching for their missing interpreter via a photograph in their possession. The player must show the image to a cooperative, local civilian, who will then recognise the person in the photograph and point the player in the direction of the interpreter. Once the player knows the location, the scenario is successfully completed. If the civilian is uncooperative, he will not respond to the player's pleas for help, and if the player is offensive or breaks the social, cultural norms [Evans, ming], the NPCs will leave. Thus, the scenario will end unsuccessfully.

This scenario was chosen because it exercises a broad range of capabilities of interactive characters: player involvement, communication between NPC and player, multiple NPCs with different attitudes, physical objects and multiple outcomes of the scenario. The scenario was also simple enough so that each team was able to reach a pseudo-code state of completing their design in a reasonable amount of time (1–3 hours). While the original IMMERSE scenario required non-verbal communication (gesture and face expression recognition), that limitation was not enforced on other systems. The specifications of the scenario were designed to be loose enough to allow each system to encode the scenario to their system's advantages without demanding external features that all systems may not possess.

3.6 Case Studies

The following three programmer-author teams of one, two, and five interview participants were studied (although there are more developers on each team). Each system was developed using different ideologies, which will become apparent in the discussion of their individual SSS. The following case studies are listed in order of increasing complexity of the modelled systems.

3.6.1 Case Study 1: BOD using POSH

Bryson [2001]; Gaudl et al. [2013] follow a particular behaviour authoring methodology entitled Behaviour-Oriented Design, described in Chapter 2.2.9. BOD combines Object-

Oriented Design and Behaviour-Based AI [Brooks, 1986] in combination with their action selection mechanism, the planner to construct agents based on their development process. BOD focuses on simplicity and iteration, offering a low barrier to entry for novice authors. This case study encoded the Lost Interpreter scenario in the least amount of time.

After a scenario is defined, a programmer and designer work together o create a list of abstract behaviours that need to be performed. It is important to note that there is no need for a BOD designer to encounter anything more complicated than interfaces and visual plan structures in their interaction with the system. This allows the designer and programmer to be the most independent of the three case studies (although they may be the same person in some projects) [Bryson and Thórisson, 2000]. In this test case, the abstract behaviour list included seven actions, including a greeting/goodbye to mark the beginning and end of the interactions, accepting, examining, returning an item, ignoring the player (for the uncooperative agent), and telling information. The second step in the process is to build what is ultimately a list of procedure signatures for the programmer, determining which of these behaviour elements need to be represented as behaviour primitives (actions and senses), as well as an idle state should all else fail [Bryson and Stein, 2001].

The programmer then codes the actions and senses as functions in the domain specific language, which will connect to the targeted system in the future (as it is at this step required to test functionality and flow of the dynamic plan). The target system can either be an animation engine or the interface to robotic actuators; the primitive actions and senses provide an abstraction layer between the DMS and the target system (as with all systems in this case study). In parallel, the designer can use the primitives (actions and sensors) created by the programmer to design the dynamic plan using ABODE[2], a graphical design tool for Parallel-Rooted Ordered Slip-Stack Hierarchical (POSH) plans, to construct the POSH-tree.

SSS Elements

i **Start Minimally:** Even though the scenario is relatively simple, it is important to begin with a minimal number of behaviours, actions, and sensors to create a working vertical slice. The scenario began with only four primitives as a first version of the plan of the core behaviours. In both ABL and BOD, authors created empty primitive stubs in their behaviour trees to structure the experience as a

[2]The latest version of Advanced Behaviour-Oriented Design Editor (ABODE) is ABODE-STAR which was extended during this PhD to reduce the cognitive load of its users. The system is freely available at: `https://github.com/suegy/abode-star`

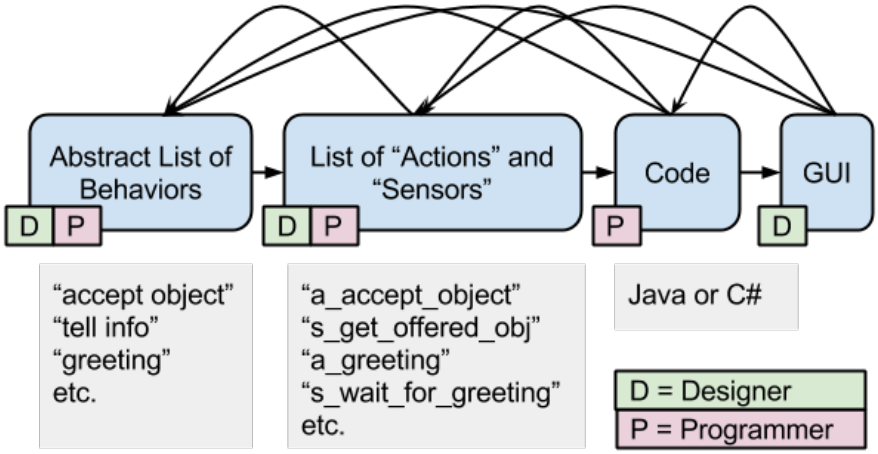


Figure 3-2: Representing the System-Specific Step for creating an IVA with Bod,FAtiMA, and Abl.

first minimal step before proceeding to fill in the stubs. Starting with a small number of action primitives supports the author when focusing on the task at hand and allows for incremental changes. It also allows the faster testing and having a robust set of actions before moving to a more advanced agent.

ii **Decompose Iteratively:** Actions only link methods from the underlying behaviour library to action nodes in the posh plan, this separation supports a more agile workflow. A key feature of the Bod authoring methodology is this agility: not only can programmers iteratively tackle the stubs created in sss Element i, but the designer and programmer freely move between the phases of the design process to build up missing primitives that were not in the minimal first list. In the case study, the programmer was creating idle and item-handling primitives while the designer realised they had not accounted for the norm−offense response. In Abl, each author's focus is on one agent at a time, and one step in the performance at a time, to systematically build the whole experience.

iii **Minimise and Encapsulate:** While not a part of this scenario, an experienced Bod designer uses the Bod metrics to keep the complexity of the plan at bay, e.g. if more than three sensors are needed to trigger a drive or competence, the logic held within the tree is getting too complex. Thus, a complex trigger should be created instead. The application of the heuristics enforces a constant minimisation of the plan which leads to a reduction in plan complexity and in turn to a process of simplifying the logic (and computational resources) controlled by the tree. Not following the metrics is a common mistake most novice Bod/posh

authors make, resulting in a tangled mess of restricting sensors that are difficult to debug and behaviour libraries limited to a narrow subset of scenarios. This last SSS element is the most unique when comparing against other approaches.

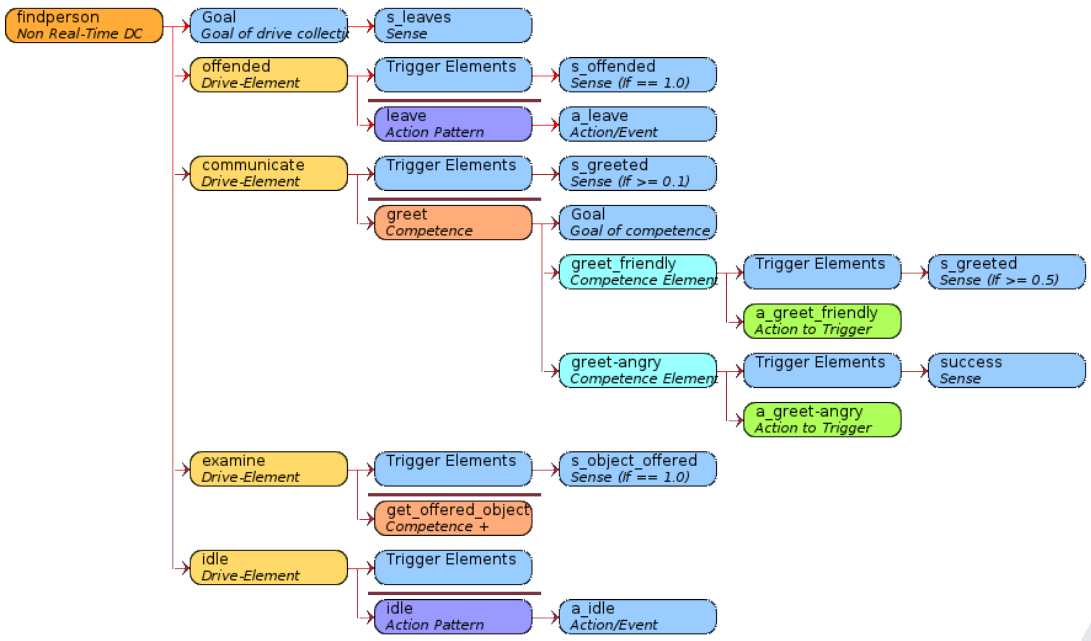


Figure 3-3: A POSH plan developed during the interviews and encoded using ABODE-STAR.

Authoring Support Strategies

The BOD/POSH case study is unique in that it is the only system with an explicit authoring approach as well as a graphical design tool, (ABODE-STAR). Due to the shallow learning curve, getting the system to work is not difficult, but creating and maintaining complex agents provides challenges in need of more robust tools. Thus, the focus of the authoring support strategies will primarily address SSS Element iii, as the first two are well-supported via the BOD methodology and the current architecture.

There is no integration of testing and debugging approach for BOD and POSH, a problem that all the other architectures which have been analysed in this chapter also share. Support for syntax checking and live behaviour debugging would shorten the programmer's development cycle considerably while iterating on more challenging behaviours (SSS Element ii). Most crucial, however, is a mechanism to facilitate better behaviour sharing and reuse between and within projects. The larger a BOD and POSH behaviour library is, the more likely that novice users tend to develop their own library

instead of reusing existing components. This produces large amounts of redundant code which results in a degrading quality of the system at some point based on the cognitive overload it requires to grasp the whole library. A new module that manages existing and comparable, encapsulated behaviour libraries, and prompts users to submit their new simplified behaviours for future reuse, would also increase the reuse and power of BOD and POSH enormously.

3.6.2 Case Study 2: FAtiMA

FATIMA [Dias and Paiva, 2011] is a new multi-agent architecture in which each agent has an emotional state and plans about the future towards a specific goal. A process which can be weighted according to the relative importance that plans and emotional state have. Different characters can have separate personality files in which these weights are defined. Authoring in FATIMA is done by editing several separate XML files. To understand the difference between ABL and FATIMA, Gomes and Jhala [2013] gives a comparison of the two systems regarding the expressiveness for modelling conflict between characters.

When presented with the requirements of the Lost Interpreter scenario, interviewed FATIMA authors started by considering the motivations of the NPCs. Since the behaviours of agents in FATIMA are goal driven, the experts proposed that an NPC must be motivated by an altruistic goal to decide to help the player. A possible example of such a goal would be:

```
<ActivePursuitGoal name="Help([character])">
    <PreConditions>
        <RecentEvent occurred="True" subject="[character]"
        action="RequestHelp" target="[SELF]" />
        <Property name="[target](isPerson)"
        operator="=" value="True"/>
    </PreConditions>
    <SuccessConditions>
        <Property name="[character](wasHelped)"
        operator="=" value="True"/>
    </SuccessConditions>
    <FailureConditions></FailureConditions>
</ActivePursuitGoal>
```

Figure 3-4: The ActivePursuitGoal specifies the conditions which are required by the agent to trigger that the agent wasHelped.

Additionally, it was pointed out that there needed to be a motivation not to help, in order to model the uncooperative NPC's behaviour. The interviewed authors chose to represent the uncooperative NPC as fearing harm from the armed player as a separate goal:

```
<InterestGoal name="ProtectSelf()">
<ProtectionConstraints>
<Property operator="="
name="[SELF](harmed)"
value="False"/>
    </ProtectionConstraints>
</InterestGoal>
```

Figure 3-5: The InterestGoals specifies if a character should protect itself based on if he or she was harmed.

For the goal to be useful, there must be an NPC action that is required for helping behaviour, but at the same time might lead it in harms way. For instance, the NPC might consider the possibility of being harmed when taking the picture from the player.

```
<Action name="Take-from-Military([military],[object])">
<PreConditions>
<Property name="[AGENT](whithInReach,[object])"
operator="=" value="True"/>
<Property name="[military](isMilitary)"
operator="=" value="True"/>
</PreConditions>
    <Effects>
        <Effect probability="1.0">
<Property name="[AGENT](has,[object])"
operator="=" value="True"/>
</Effect>
<Effect probability="0.1">
<Property name="[AGENT](harmed)"
operator="=" value="True"/>
</Effect>
    </Effects>
</Action>
```

Figure 3-6: Take-from-Military specifies more complex conditions which need to be met before the characters takes an object form the player.

Thus, if the agent considers a plan in which they may be harmed, it will trigger a Fear emotion. Next, the authors defined actions that the agents could take along the path to reaching the help goal, such as taking the photo, examining it, or speaking, which is where the authoring process in BOD and ABL began.

SSS Elements

Based on a second FAtiMA authoring team further insights into its underlying process were gained after iteratively discovering the SSS elements with the first team. Their responses have been included in the following sections alongside those of the original FAtiMA scenario authoring team.

iv **Goals First:** FAtiMA goal primitives must be defined first, with the necessary actions being derived from them. This is driven by FAtiMA's dependence on goals for the cognitive appraisal emotion model to work. For each branching strategy that the agent could take (respond to request/ not respond), there needed to be a motivation, hence a driving goal. The second FAtiMA team worked with goals and actions simultaneously, which was inconsistent with the first team. Part of the second team's reasoning was that in planning, with the appropriate set of actions, the agent should be able to deal with a wide range of situations, and thus goals. It can be speculated that this different approach may be related to the different authoring experience and scenario complexity between both teams.

v **Find Decision Points:** During the interviews it was noticeable that the interviewed authors divided the scenario into sections whose boundaries corresponded to moments in which the civilians had to make a decision. As every decision point must also be motivated by a goal. This approach helped to author the previous SSS element as well. Authors also found that temporal ordering of decisions could be enforced by creating goal preconditions that referenced recent events. The second FAtiMA team agreed with the analysis of the first team. Thinking the decision point sequences through helped define goals for both teams.

vi **Goal Weighting and Tuning:** The cooperative and uncooperative civilians in the scenario choose to take different actions when deciding to help. This decision process was made by using different numerical weights for the Help([character]) and ProtectSelf () goals. By giving more importance to a particular goal in the character's personality file, the interviewed authors made sure that each agent made the appropriate decision at the previously described decision points. These goal weights completely control how different agents take different paths throughout the performance. This process supports previous comments by FAtiMA authors (including the second FAtiMA team) that weight tuning is by far the most time-consuming process of complex FAtiMA authoring Bernardini and Porayska-Pomsta [2013].

vii **Intent Goals for Future Consequences:** While not part of this particular authoring scenario, a useful authoring anecdote was encountered that sparked discussion of this additional FAtiMA sss. Goals have two types: ActivePursuit and Intent. For ActivePursuit goals, the agent creates plans to achieve them. Intent goals define constraints that should try to be enforced. In the process of understanding those approaches [Gomes and Jhala, 2013], one author tried to create two ActivePursuit goals that an agent simultaneously attempted to achieve. However, after referencing an expert FAtiMA author, the author found that FAtiMA is only able to pursue one ActivePursuit goal at a time. This initial misinterpretation of the system lead to a re-write of their entire goal structure. The second FAtiMA team did not agree that this was an important part of their process, as their authors reported to be able to choose easily between either goal types.

Authoring Support Strategies

Authoring support strategies for the two sss elements were discussed that were backed by both teams: v and vi. For sss element v, an interface is proposed where authors can create example sequences of events schematically. Afterwards, the tool is intended to prompt the user at which points a given agent has to make decisions. For each of these points, the author can create corresponding goals[3].

All three case studies have points in their authoring process where quick iterations of different scenarios would be incredibly helpful in speeding up the process. FAtiMA exhibits the most obvious case of tweaking, as all of its content adjustments can be narrowed to values in a handful of specific files. The authors speculated that launching multiple simultaneous configurations of a scenario with FAtiMA agents encoded with different personality weights (possibly in real-time), choosing the most appealing presented version, and iteratively repeating this step could demonstratively narrow down on a target.

3.6.3 Case Study 3: ABL

Abl, presented in Chapter 2.2.8, was designed with a focus on the creation of expressive IVAs and provides a reactive planning language for structuring and creating them with a high degree of interactivity [Mateas and Stern, 2002]. The primary structure primitive in Abl is the behaviour, which can sub-goal other behaviours in sequence or in parallel and contains preconditions that gate whether or not it can currently be executed. The Active Behaviour Tree (ABT) encodes the current intention structure of the agent,

[3]The author could also add possible actions, which based on sss element iv would motivate different agent strategies.

with the leaves of the tree as candidate, executable steps. Working Memory Elements (WMEs) hold information intended to be shared throughout the ABT, such as whether an NPC is holding an item. It is important to note, all the interviewed ABL authors are involved with the IMMERSE project, and ABL language-specific and IMMERSE project-specific constructs will be specified. BOD and FATIMA offload this extra layer of control, but it represents the core of how all the abstract behaviours outlined earlier were encoded in ABL.

As with BOD, the ABL authors approach a scenario by first creating a list of abstract behaviours which are stubbed into the ABT in a rough sequential structure. At a high level, the authors each tackle a specific behaviour and work iteratively with each other to bring it to completion. ABL authors thus also employ the SSS Elements i and ii described above. However, the details of the iterative steps for ABL allow for possible alternatives more focused on ABL potentially leading to further SSS elements.

For any behaviour that an NPC may direct towards a human player, there is the need of separate behaviour sub-trees, created for making the NPC perform and wait for the signal that the player has taken the action. The following demonstrates an example from the scenario, illustrating the give_object() behaviour if it was used between two NPCs (note that the give_object() behaviour assumes it was triggered by a request_object() behaviour or that the target will accept the offered object unconditionally):

```
1  sequential behavior give_object(String myName, String targetName, String
       objectName) {
2      // The precondition grabs the target's PhysicalWME
3      precondition { characterPhysicalWME = (PhysicalAgentWME)
4              (characterPhysicalWME.getId().equals(targetName)) }
5      Location characterPt;
6      SocialSignalWME socialSignal;
7      // grab the physical location of the target
8      mental_act { characterPt = characterPhysicalWME.getLocation(); }
9      // NPC offers the object it is holding
10     subgoal headTrack(myName, targetName);
11     subgoal turnToFacingPoint(myName, characterPt);
12     subgoal performAnimation(myName, targetName, animationOfferObject);
13     mental_act {socialSignal = new SocialSignalWME(
           socialInterpretationExtendHand, myName, targetName );
14     BehavingEntity.getBehavingEntity().addWME(socialSignal);}
15     // wait for the person who will take the object to set this flag
16     with (success_test { (socialSignal.getChosenInterpretation() != null)
           } ) wait;
17     // Make the photo disappear from my hand, the action of taking the
           photo sets it in the target's hand
18     act attachObject(myName, objectName, false); }
```

Figure 3-7: A sequential ABL behaviour for requesting an object.

- **The context of how the behaviour will be triggered**: In this scenario, the author knows that the request_object () behaviour triggers give_object (). It contains no logic for having the offered object rejected. This behaviour also only handles removing the object from the character's hand and assumes another behaviour handles the object's fate.

- **Relevant signals and WMEs**: The previous behaviour was authored assuming that the characterPhysicalWME contains locational information, that there is a socialSignalWME ready to handle the socialInterpretationExtendHand interpretation, and that there are constants such as the cExchangeObjectDistance previously defined and calibrated for the world. If any of these are lacking, or the author does not know about them, the author must search the existing code or create them.

- **Expected animations**: Head tracking, eye gaze, and holding out the offered object are the animations used in this behaviour. The logic behind procedurally animating them is handled elsewhere, and if it were not, the author would have to create it.

- **Possible Interruptions**: The most crucial step to making these behaviours robust is handling interruptions, which the above behaviour fails to do. In the success_test , if the NPC never acknowledges the socialSignal or the player never comes in range, the NPC will hang with their hand held out forever. If a timeout was added to holding out their hand, it is unclear what the NPC should do about the unrequited object offering, or how it should handle the lost request_object () context. These are all considerations the author must address when aiming for robust behaviours.

SSS Elements

viii **Define Coding Idioms:** Unlike Bod and FAtiMA, which make strong architectural commitments to specific agent authoring idioms, Abl is a more general reactive planning language similar to STRIPS planning described by Ghallab et al. [2004]. Within Abl different Abl idioms can be implemented such as the task managers described by McCoy and Mateas [2008]; Weber et al. [2010a] or the goal ideom by Weber et al. [2010a]. Before novice and intermediate programmers can make progress, an expert Abl programmer must first define the coding idioms used to structure the agent (see Weber et al. [2010c] for another example of Abl idioms). These idioms define approaches for organising clusters of behaviours to achieve goals. For the immerse project, the Social Interaction Units

(SIUs) idiom has been developed to organise clusters of behaviours around goals driving specific social interactions. Those goals are similar to BEATS described in Chapter 2.2.8. The interviewed ABL authors all made use of the SIU idiom when working on the "Lost Interpreter".

ix **NPC and Player Considerations:** Although the example behaviour above, as well as the architecture, is separated from a particular implementation, the code must intimately consider implementation details. There is an enormous amount of state information and ABT possibilities the author must be aware of such as how the behaviour will be triggered in the performance or whether NPC or PC characters will be performing or responding to a behaviour. BOD and FATIMA offload this extra layer of control, but it represents the core of how all the abstract behaviours outlined earlier were encoded in ABL.

x **Consider Interruptions:** In the given scenario, if the system detects the player offering the photo, it will trigger the sequence of ABL behaviours by the cooperative NPC: take_object(photo), examine_object(photo), and point_to(interpreter). If the system detects the player requesting the photo back any time after examine_object(photo), the ABT will trigger the NPC to give back the photo regardless of whether it is in the middle of another behaviour such as pointing. From a designer's perspective, it makes sense that someone may extend the photo in return with one hand and point with another. The author of point_to() must be made aware that the behaviour may have to multi-task with other behaviours and take precautions to perform it appropriately. If the synchronisation of those behaviours is not done properly, the animation of the IVA will contain artefacts which are not appealing. Blending character animations is not a trivial problem and is still a focus of animation research.

3.7 Authoring Support Strategies

ABL's SSS were discussed with novice, intermediate, and expert authors of the ABL approach, and their processes all shared the same structure described in detail above. However, novices and early intermediate authors *needed* expert guidance to understand that the above considerations existed, where to look for them in the code or how to create aspects of them if they were missing. Once example behaviours have been created, authors routinely copy-paste huge sections of code. This process is highly similar to the approaches of professional developers when utilising advanced game AI techniques such as artificial neural networks, potential fields or as the most prominent

example A*; they initially use those textbook examples and, later on, modify them according to their needs. Industrial publications contain typically large parts of source code examples going beyond simple pseudo code for exactly the same reason.

In contrast to the visual representation of BOD's dynamic plan, the Active Behaviour Tree (ABT) in ABL is in constant flux, making it hard to visualise without presenting just a non-representative snapshot. The main reason for a changing ABT is the possibility of spawning new goals or the ability of the planner to add change the current behaviour to better match the current goal. This approach is similar to GOAP by Orkin [2005] and presents similar challenges to authors, increased cognitive load when designing behaviours. Thus, Orkin offers designers only the option to add locations and specific goals for those to reduce the design complexity which is not sufficient for ABL authors. Currently, ABL authors use debug log print statements of the current system state and *trial-and-error experiments* to determine the correctness of their implementation which is insufficient for complex agents as some behaviours might be only visible at specific points in time. More sophisticated debugging techniques exist in the form of an ABL debugger (a process that executes alongside the ABT at run-time), but the debugger was not used by any participant.

Expert authors report the debugger to be unstable, hard to use and set up. This indicates possible reasons for why it has not been used further. The debugger was developed by Weber et al. [2010b] during an ABL STARCRAFT agent project. The debugger requires the developer to attach the debugging process to the currently active agent which is a non-trivial task, requiring special privileges on the computer. Therefore, the initial step to employ the debugger is already a hurdle for novice users.

BOD authors using JYPOSH have a working behaviour tree editor ABODE supporting the design of behaviours by presenting the agent tree during development visually. However, ABODE does not provide inspection during run-time which reduces its usefulness after the POSH plan is developed. Additionally, JYPOSH requires a complex setup which confused novice users. Similar to ABL, BOD and POSH do not provide debugging support or any designed feedback during the execution of the agent. Novice developers included debug statements into the underlying behaviour library to trace the execution of primitives. Another negative point identified by novice users is the handling of errors. Due to the usage of bindings between Java and Python within the arbitration process, JYPOSH returns cryptic error statements and memory dumps which are unusable by non-expert developers. A better feedback and debugging support are essential in industrial applications as described in Chapter 2.1.1 where each tree node in a BT returns a statement of success, fail, or error and the tree can be visualised showing that state.

3.8 Summarising the System-Specific Step

In this chapter, the SSS requirements methodology is proposed as a means by which programmer-authors may better understand their IVA architecture's authoring burden and make progress toward alleviating it. The methodology was evolved through a series of interviews conducted with a set of disparate and independent groups performing IVA authoring research. After that, case studies of three teams authoring a single simple scenario were performed where their authoring process was process-mapped, extracted and elaborated. As part of this, their SSS and its elements, in combination with proposed authoring strategies that might alleviate their authoring burdens were analysed. As a result, the three teams found the SSS to be a valuable tool in analysing their system, and each group plans on implementing their previously proposed authoring strategies.

In the appendix C.1 an additional overview is provided which presents a reduced presentation of all SSS elements found across the case studies. Instead of looking at all SSS in we will focus on a sub-set affecting BOD which will be discussed now.

Although the SSS concept contains the phrase "System-Specific" in its name, it was found that certain SSS elements are shared between certain systems. A starting hypothesis of this work was that not all IVA authoring architectures are completely isolated from another, and based on the similarity and overlap of the SSS this hypothesis could be supported by the findings.

This hypothesis is additionally backed by the structural similarities of ACT-R, SOAR and ICARUS discussed in Chapter 2.2.5. Those three architectures for modelling cognitive agents are based on the same principle of a unified theory of cognition and even though they are differently implemented, they share common structural elements such as central places for long and short-term memory or modules for perception which interact with the memory. It seems this is also true for most IVA systems which encode similar implicit rules for designing agents and vary only for certain specific aspects of the design. We believe that the SSS approach not only aids other architectures to discover their individual SSS elements but that those architectures are able to reuse the SSS elements and the corresponding authoring support strategies that have been outlined.

For BOD the SSS *i* to *iii* are explicitly given by Bryson [2000b], supporting authors more than other approaches when approaching the design of new agents. This explicit guideline is supporting the initial learning curve of novice users and strengthening their understanding of the design steps. However, the SSS v and x are not explicitly given in the approach and were identified using the SSS method. Nonetheless, the

design approach for BOD states that a high-level task should be specified initially, which is then decomposed, thus implicitly encoding SSS v. However, novice developers tend to focus on reaching local goals rather than global goals as demonstrated by Partington and Bryson [2005]. Partington and Bryson [2005] specify goals only for individual sub-tree which ignores more global concerns, whereas ABL authors have to focus on global and local goals at the same time. The ABL approach requires a global understanding of the situation which, for complex agents, is demanding but essential for developing sophisticated IVA. For BOD a local focus is initially enough for developing the first iterations of the agent but is essential for moving towards more sophisticated agents. Concerning SSS x, interrupts are in ABL handled by the planner to some degree. Handling interrupts is essential in avoiding behaviour dithering, a common problem in dynamic systems and agents. Rohlfshagen and Bryson [2010] present an approach for BOD to handle dithering using a method that generalises to different scenarios but there is no mechanism on a design level to support it.

Generally, BOD does provide more support when designing light-weight agents in comparison to ABL and FATIMA, but misses advanced functionality to support a more robust development of more complex agents. To write plans which are of similar size to ABL plans, editing support is needed similar to those of Integrated Development Environments (IDES) for programming languages. Once an agent in BOD reaches a certain complexity debugging support, robust handling of errors and the underlying library becomes essential. The setup of JYPOSH presents a hurdle to novice users as it requires special privileges. For larger projects, such as IMMERSE, teams of developers have to work on shared tasks including the design of IVAs, BOD currently does not support the distribution of work due to its light-weight nature but further improvements could address this. In Chapter 5 a novel approach based on BOD and POSH is presented which addresses the discussed issues and aims at addressing the need for a more robust design method by providing mechanisms that support team collaboration and task sharing as well as robust behaviour development.

The next chapter demonstrates the application of a light-weight architecture to an extremely complex problem domain—REAL-TIME STRATEGY (RTS) games. This demanding domain requires planning on multiple levels of abstraction maintaining long-term goals and reacting to changes in real-time. This case study is a proof of concept for using light-weight approaches in such domains and is essential to understanding limitations and requirements for agent design architectures. In addition to the case study, the chapter also presents an approach to integrate existing encoded human knowledge into the underlying logic of game agents, offering a new approach for developing sophisticated agents without the need for programming.

Chapter 4

Integrating Human Knowledge into Game AI

In the previous chapter, we examined three Interactive Virtual Agent (IVA) frameworks, Behaviour-Oriented Design (BOD), **F**earNot! **A**ffec**ti**ve **M**ind **A**rchitecture (FATIMA) and ABL. Based on a series of interviews with developing teams of each platform, unifying elements in the three approaches were identified and the SYSTEM-SPECIFIC STEP (SSS) methodology for identifying requirements has been proposed. The SSS allows a developer to identify the implicit assumptions and process underlying their framework. Once brought to light, this knowledge aids the understanding of weaknesses in the initial process and affects the learning curve for novice developers positively due to the extraction of now explicit rules of development.

In this chapter, we will discuss one case study for developing agents for a highly demanding AI domain—REAL-TIME STRATEGY (RTS). The case study will highlight the possibility of using a light-weight architecture in combination with a development methodology to develop sophisticated agents in a robust manner. An approach of how to build agents based on existing encoded strategies in game forums is discussed next, presenting details towards a more complex agent that implements this knowledge in a form resembling the forum notation. This demonstrates the inclusion of expert knowledge from non-programmers into the core part of the game logic with the aim to allow even novice programmers to develop sophisticated game agents.

4.1 Contribution

This chapter is based on a paper presented at FDG2013 by Gaudl et al. [2013]. Davies [2012] developed an initial base for the agent design of the discussed STARCRAFT agent

as part of his final bachelor project. The initial agent design is complemented by my literature review as well as my own analysis and extension of the initial design of the agent. The chapter presents a case study of developing a complex StarCraft agent and a further proof of concept extension to it. During a period in 2012, the work conducted by me and Simon Davies on the paper was extending his original approach to be able to compete at the annual StarCraft competition run by the Computational Intelligence in Games (CIG)[1]. This project supported the present chapter and the overall understanding of complex StarCraft agents. I then derived design steps based on Bod that can be captured using sss which was introduced in the last chapter. After the initial project was concluded, I extended the work further, developing a template agent and interface layer for BroodWars API (bwapi) and StarCraft. Thereby, I extended the initial design/approach and altered it drastically and developed a new way of integrating hand-authored strategies which are available for proven strategies. My contribution to this work is 80%.

4.2 Problem Description

For most tasks or problems in real life and in the virtual one, the computationally or cognitive most expensive part of completing a task successfully is the search process of finding an optimal or at least sufficiently good solution. This process is sometimes combined with an additional criterion, finding the right solution in a given time.

In nature, it often is not beneficial to come up with solutions which are too late because the penalties for deciding late are quite often drastic. For example, an antelope is deciding which direction to take to escape from a predator. If the antelope is waiting too long, it will simply be too late to escape, and it will die. Even though this happens quite often in nature, it is not in the interest of the animal that is trying to escape. For some artificial systems, similar restrictions to finding a sufficient solution apply, e.g. collision detection or avoidance systems need to discover solutions before a collision happens.

This time restriction means, that the Decision-Making System (dms) needs to be aware and able to handle and scale to restrictions such as time-bound solutions. Natural agents normally do this implicitly, however, modelling a system to take dynamic time constraints into account is quite complex.

The actual application of the solution to the problem is typically less expensive. This is true under the assumption that the execution of the solution does not require extra computational or cognitive resources, e.g. fine motor skills or complex, intricate

[1] `http://www.ieee-cig.org/`

action sequences. If those are needed, the problem can be treated as a second-order problem which needs an additional meta-solution, the integration of the lower level solution for the execution.

How to reduce the search time? One obvious answer to that question is to outsource or pre-compute the search. This answer initially sounds like nothing more than a cheap trick but in most cases where the search space is vast, including external knowledge is beneficial.

When picking a domain such as games, for games with a low impact of random events, such as Noughts and Crosses or Go–played on smaller game boards—a winning strategy exists and is known [Müller, 2002]. Which means, that the intense computational task of searching through the entire game space to find a trajectory from the current state to a good solution is solved. Thus, the best possible choice at each step can be taken by the system by calculating a path along that trajectory. This strategy can then be programmed into an AI system controlling an artificial player. By including this knowledge derived by an external mechanism, the actual time to find the solution is negligible. For Noughts and Crosses most adult players can fully understand and solve the game which results in most of the cases in a draw game. The resulting experience is most likely less entertaining once the strategy is known.

For more complex games, however, the game space is not fully known or explored and winning strategies are not known or do not exist. For chess, it is currently believed that a winning strategy exists, but the strategy is not known. For some of those complex games, we do however have large collections of recorded games. In chess, some of those are sorted and selected for specific collections referred to as opening books for the start of a game or end game books for the opposite. Those books are simply put an encoded form of recorded sets of plays from a particular state to another given state. They are used in chess to help predict a possible outcome of a game. To reduce the game space in chess, a player applies the knowledge from an opening book to direct the game towards a more favourable state. The navigation between the states is done by comparing the current state of the board to existing states in the book and selecting moves which are on a path to the desired state. The knowledge of those books is essential–if not crucial–to chess. Most if not all advanced chess players apply it to their playing. By doing so, they can reduce the risk of running into a game state which is less favourable. This also shifts the original tactical play of chess from the board level, which essentially is the ability to predict possible moves of the opponent for multiple moves, onto a meta-level of knowing the more current or larger book. For chess AI, the size of the available opening book and its contained games plus the planning depth between transitions of

existing games determines how good the system is [van der Werf et al., 2003; Müller, 2002]. This defines depth as the number of predictions of possible opponent moves and their appropriate responses. If the chess AI is scaled to a particular user, most of the time those parameters are modified to achieve a level of skill compatible with the competing human player.

Before the recent success of chess AI in the early 2000's, it was often argued that it is possible to exploit a weakness in the size of the opening book by making a move in a chess game which the computer did not know–a move with a piece that was not contained in its opening book. This exploit was considered potentially beneficial; it allowed the player to get the upper hand on the computer. However, taking into account those recent successes in the chess competitions and the amount of skill it needs to achieve even a draw let alone win, this argument now no longer holds. What remains, in that case, are two hard problems. The first of which is efficiently searching within the space confined in the opening books for openings with a state similar or preferably identical to the one on the board. The second is, proposing a chess move leading to the desired outcome either by picking one of the found games in the opening book and guiding the current match or by calculating a new move leading back to a desired contained game in the opening book. These problems by themselves are computationally quite demanding but not part of this work. However, the idea of creating a good artificial player by employing encoded pre-existing gameplay or knowledge about the game into the AI system will be the primary focus for the remainder of this chapter and a large focus of this thesis.

Moving from analogue games like Noughts & Crosses, chess or GO, to digital games additionally changes the problem space in some ways. Not only are digital games a lot younger, which means that the space of existing successful strategies is less explored. Most of the time, they are also even more complex than traditional board games bringing in elements of real-time interaction, large-scale game boards, a larger chance impact or a multitude of different game pieces. However, digital games bear one similarity to traditional board games. It is most of the times possible to encode human knowledge into the AI system of a game which is similar to the existing opening books in chess. By doing so, it is possible in the same way as for chess to reduce the computational time for finding a good move. For most games such strategy collections or opening books are not recorded explicitly either because the games contain too much freedom, making it hard to specify a strategy, or the games never achieved sufficient popularity.

Real-Time Strategy RTS games are a sub domain of digital games.They are quite similar to traditional tactical board games such as chess. A prominent game in this genre is StarCraft which was released in 1998 by Blizzard Entertainment[2]. It attracted a huge player community due to its well-balanced, complex gameplay and its focus on tactical and skill-based play. StarCraft, focuses on strategic real-time player-versus-player (PvP) gameplay in a futuristic setting. Due to the good balance of the available in-game parties the players can choose from, it has become famous in e-sports [Taylor, 2012] and attracted significant media attention.

The game provides a complex environment with many interaction possibilities; those involve players having to manage a multitude of different tasks simultaneously. StarCraft also requires players to be able to react to changes in the game world in real-time while in parallel controlling and keeping track of long-term goals. This forces the player to differentiate between micro-management of short term goals and units while maintaining long-term management of an advantage over the other players. The given setting introduces many challenges regarding pro and re-activeness of an agent, planning, and abstraction, modularity and robustness of game AI implementations. Due to StarCraft's popularity in the player community, extensive collections of strategies and recorded matches are available online. Based on the skill-based gameplay required by players and its popularity in the media, StarCraft created continued interest spreading into the research community which lead annual competitions on StarCraft at major AI/CI conferences and meetings.

4.3 StarCraft AI Design

In this chapter, the possibility of offloading the computational or cognitive expensive task of finding a good trajectory through the solution space of possible approaches to a problem will be evaluated. To research the feasibility of this idea the presented case study moves away from the encoded knowledge in opening books used in chess to encoding human knowledge in digital games, in this case, StarCraft.

The work builds upon the planning and agent design approaches presented in Chapter 2.2 and extends on the previous hypothesis that the most crucial task in finding a good solution to a problem is narrowing down the search space. Let us refine it further to account for digital games:

> "It is possible to create a flexible, well-performing game AI agent only by utilising existing human player strategies on top of a basic behaviour layer."

[2] `http://us.blizzard.com/en-us/games/sc/`

Behaviour-Oriented Design was chosen for the agent design, described in detail in Chapter 2.2.9 because of its modular and light-weight architecture and the design support it offers when creating new agents. Unlike approaches that attempt to use a single method for all of the agent's behaviour (e.g. Potential Fields, see Chapter 2.1.2 or FSMs see Chapter 2.1.1), BOD fosters the usage of specialised low-level approaches and combines them using the high-level planning system.

Supporting human design when developing an AI system is an important task because in most of those systems human designers and software engineers are still required to do most of the hard work when coping with the combinatorial explosion of available options [Bryson and Stein, 2001]. Additionally, the iterative design heuristics provided by BOD aid developers to find a more intuitive and better way to solve a problem, whether with their modules or by integrating external ones responsible for activities such as learning or black-box approaches such as Neural Networks. A more in-depth description of BOD is given in Chapter 2.2.9.

4.4 Related Work

STARCRAFT has been a centre of research attention for nearly a decade now and there are numerous publications using it to test, develop and evaluate AI approaches. Most of the research was motivated by the repeated "Call to Arms" of Buro [2003]; Buro and Churchill [2012]. In those calls, Buro motivates that RTS games offer an ideal testbed for developing different AI approaches while testing within complex, demanding environments which focused on STARCRAFT after the release of BWAPI[3].

The most similar approach to the one presented in this chapter is Goal-Driven Autonomy used with ABL by Weber et al. [2010a]. Weber's approach is to some extent based on earlier work for another real-time strategy game [McCoy and Mateas, 2008], WARGUS[4]. Weber's approach models different managers for high-level tasks in ABL and uses a Java wrapper in combination with BWAPI to implement the agent. As discussed in Chapter 2.2.8, ABL uses a reactive planner to achieve different agent goals by chaining them together based on their preconditions. Using this approach it is possible to create different behaviours which at run-time are able to interact with each other, thus, generating the final behaviour expression of an agent. The resulting STARCRAFT

[3] BWAPI offers an interface to access and change data within StarCraft. Thereby, it allows the inclusion of external code to represent artificial agents within the game. The API allows two modes which either offer the inclusion of an agent through a TCP/IP connection or through direct memory access when using a dynamic library file ("DLL"). More information are available at: `https://github.com/bwapi/bwapi`

[4] WARGUS is real-time strategy game based on the mechanics of the game WarCraft by Blizzard. More details are available at: `http://wargus.sourceforge.net`

agent performs well and was able to compete with novice human players. By separating different activities into different managers, Weber can split the required focus on the whole agent and the task at hand into smaller pieces. This de-coupling of different tasks reduces the complexity of the agent. He implements each of those managers as parallel behavior components which pursue individual goals. The whole agent is hand-coded but the resulting behaviour emerges through the recombination done by the planner. During this work, Weber developed a debug-tree visualiser to manage and understand the ABL behaviour tree. As the design was programmer-driven, this indicates and supports the hypothesis that tool support and visual representation are important for complex agent design.

Other approaches such as EVOLUTIONARY POTENTIAL FIELDS (EMAPF) [Hagelbäck, 2012; Hagelbäck and Johansson, 2008], offer a different perspective on agent design by using a force-based model to attract or repulse units; a more thorough discussion of potential fields is given in the literature review, Chapter 2.1.2. In their work, Hagelbäck and Johansson use a hard-coded build strategy for the agent derived by experimentation and focuses on dynamic unit control. Through the use of potential fields, they are able to dynamically respond to the opponent in a fluent and visually understandable way. In their implementation, each opponent unit has an attractive force based on its benefit to the player's score. The utility is based on the unit health and its potential harm to the player. Once a unit controlled by the PF has fired the polarity of the attractive force changes and repulses. This creates a wave-like movement of the PF units which resembles schooling fish [Couzin et al., 2011] to a degree. Using PFs, however, requires parameter tuning for a multitude of gravitational forces for which they used an evolutionary approach. Evolutionary approaches offer, similar to artificial neural networks, a robust but time-consuming way for parameter adjustment. In cases where training time and data is available they exceed other approaches, however, training time and data are normally limited in games and especially in early game development, see Chapter 1.2.1.

Another approach which is gaining more attention in recent years is MONTE-CARLO TREESEARCH. Soemers [2014] developed an agent for STARCRAFT using MONTE-CARLO TREESEARCH (MCTS) to select appropriate moves. MCTS is using random sampling in combination with simulated annealing, see Chapter 2.1.3. Similar to the previously discussed EMAPF, the downside of using any evolutionary approach applies to MCTS as well. However due to the simulated annealing and the usage of a special cost function (UCT [Browne et al., 2012]) , it is possible to reduce the number of simulations and only pursue moves which are beneficial to the agent's utility. Due to the large state-space of possible moves an agent can make the approach is not applicable

for any live competition before learning.

There also exist earlier works on RTS games before the community moved to StarCraft such as Sorts by Wintermute et al. [2007] or work on Wargus by McCoy and Mateas [2008]. In their work, Wintermute et al. implement artificial agents for Open Real-Time Strategy (ORTS)[5] using the SOAR cognitive architecture covered in Chapter 2.2.5. For their implementation they combine Finite-State Machine s (FSMs) for controlling low-level behaviours and only use SOAR on a high abstraction level to arbitrate between a small subset of goals. The implementation of all behaviours, as with the ABL agent, is hand-authored and in the case of Sorts the production rules were designed to follow simple strategic plans. However, the usage of ORTS as a research tool brought issues with itself which are discussed by Wintermute et al. [2007], namely the full access in the game sources resorted in the agent was being able to crash or interfere with other agents, making competitions more problematic than beneficial.

4.5 Case Study: BOD applied to Real-Time Strategy Games

In this case study, the feasibility of developing a light-weight cognitive agent will be demonstrated by following the BOD design methodology to build a modular agent for a highly demanding Real-Time Strategy game environment. Thereby the key aspects of BOD will be demonstrated, namely the focus on object-oriented behaviour design, reusability and a well-guided approach to more robust agent design. BOD has been previously used to create agents for by Partington and Bryson [2005] to demonstrate that the approach can handle real-time control in a fast-paced 3D game.

4.5.1 StarCraft

For this case study, a first prototype AI for StarCraft was designed to play the Hivemind of the Zerg, one of three different races of the game. It is pitted against a variety of other AI systems including the commercial agent the game was originally shipped with and several available AIs over the Internet.

To aid the understanding of the problem space, we first take at closer look at the game. Figure 4-1 illustrates the player's view of the game for a Zerg player. The screenshot shows the starting position right after beginning a match. A mini-map is given on the lower left which allows the player to quickly access locations or keep a general overview of the game. The agent has access to the same amount of information

[5] ORTS is a programming environment for developing real-time strategy AI and testing it. It is open-source and allows offline or online game sessions. More information are available at: `https://skatgame.net/mburo/orts`

Figure 4-1: The StarCraft interface showing the central building of the Zerg race—the Hatchery. The area surrounding the Hatchery is covered with creep offering additional bonuses to Zerg units of the controlling player. The crystals on the left side are one of the resources the player can gather. Next to the crystals are four units of the player—drones—which are responsible for gathering and building. The flying unit to the right—the Overlord—is responsible for scouting, detecting hidden enemies and building up supply.

and is able to control, same as the player, each unit separately. Controlling individual agents and shifting their focus is an important skill of professional players. Approaches such as EMAPF handle individual assignments with ease in contrast to human players. However, even mediocre human players are still better than any current AI approach at handling multiple levels of abstraction, control and strategic decisions. This difference in ability when comparing mediocre players and well advanced AI techniques renders StarCraft one of the most challenging areas of game AI research.

The related work in Section 4.4 highlighted similar approaches to problems within the field of game AI and for StarCraft in particular which allows us to position Bod in relation to them. While machine learning techniques and evolutionary methods can be powerful tools, they cannot solve large problems in tractable amounts of time. For StarCraft, those approaches require more computational resources or training than initially available, making their inclusion in the commercial environment more challenging. Additionally, Bod focuses on a design methodology incrementally working on hand-authored plans. This makes the Bod approach similar to Abl or the SOAR

agent. It allows the developer greater control over the resulting behaviour of an AI system. But in contrast to ABL a visual designer can be used—Advanced Behaviour-Oriented Design Editor (ABODE)—in combination with a stronger separation of plans and underlying implementation which Orkin [2005]; Hecker [2009] argue is an important feature to aid the understanding of the whole agent.

4.5.2 System Architecture

To develop an agent, the first task is to set up a domain specific agent system by integrating Parallel-Rooted Ordered Slip-Stack Hierarchical (POSH) into the game. The resulting agent will utilise a basic high-level strategy similar to the one presented in Figure 4-2 using ABODE.

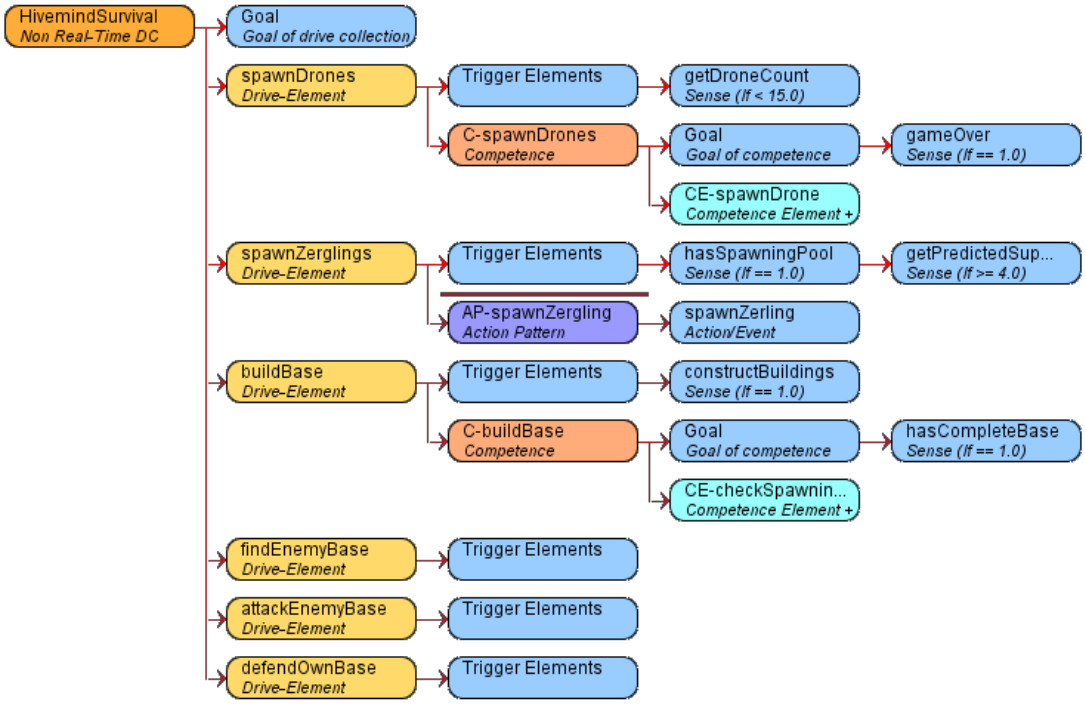


Figure 4-2: Initial POSH plan from inside the ABODE Editor for the Zerg Hivemind which builds drones, mines Crystal and builds Zerlings after creating a *SpawningPool*.

The system comprises of the game in client-server mode which uses BWAPI[6] on the server side to communicate to BWAPI client connected to the agent, see Figure 4-3.

[6] BWAPI offers an interface to access and change data within StarCraft. Thereby, it allows the inclusion of external code to represent artificial agents within the game. The API allows two modes which either offer the inclusion of an agent through a TCP/IP connection or through direct memory access when using a dynamic library file ("DLL"). More information are available at: `https://github.com/bwapi/bwapi`

To create the BOD agent, JYPOSH is used, a jython implementation of the Behaviour-Oriented Design action selection mechanism. BWAPI is written in $C++$, to access it the JAVA NATIVE INTERFACE wrapper for BWAPI called JNIBWAPI is utilised to communicate to the game. The result of using JNIBWAPI in combination with JYPOSH is that the behaviour libraries can be written in Java and Python.

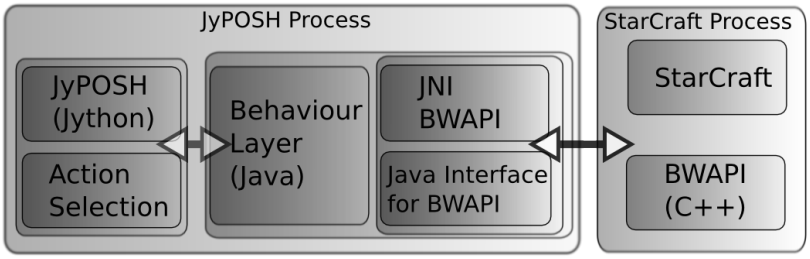


Figure 4-3: The architecture model for the StarCraft BOD AI. The architecture is separated into three elements the action selection done by POSH on the left. The middle part represents the behaviour primitives, e.g. actions and senses. The rightmost block shows the actual game and the interface which allows the deployment of strategies.

These new behaviours can then be linked to the planner which arbitrates between the behaviours and selects appropriate actions based on the state of the plan. The overall architecture is split into two processes which run independently. The process which contains the game and BWAPI is advancing the game at 25Frames per Second (fps) and communicates asynchronously with the agent. The agent is formed in the second process which contains JNIBWAPI as a communication layer, the behaviour layer which contains the POSH action primitives which are called from the action selection module of JYPOSH as shown in Figure 4-3. The behaviour layer contains the short term memory of the agent responsible for tracking the state changes within the game. The action selection module is formed once the planner loads the POSH plan and advances it each cycle.

Once the system architecture had been laid out, the next step was to create a basic agent that performed most of the basic tasks required to play a game of StarCraft. To do this, the first goal was to implement an agent that would only build the basic offensive unit, the Zergling, as quickly as possible, and then attack. This is a basic strategy that has been used by players to attempt to win games quickly without having to have a long-term strategy, see Figure 4-6.

JNIBWAPI makes use of the capabilities of BWAPI where it acts as a proxy. Here BWAPI runs as a server, and the AI connects to this to control STARCRAFT. This allows the flexibility to run external applications such as JYPOSH. An alternative approach would have been developing the agent as a dynamic library which can be

```
/** Builds a pair of zerglings  from any larvae**/
public boolean spawnZergling(){
  for (Unit unit : bwapi.getMyUnits()) {
    if (unit.getTypeID() == UnitTypes.Zerg_Larva.ordinal()) {
      if (resourceManager.getMineralCount() >= 50 && resourceManager.
          getSupplyAvailable() >= 1 && buildingManager.hasSpawningPool(
          true)){
        bwapi.morph(unit.getID(), UnitTypes.Zerg_Zergling.ordinal());
        return true;
        }
      }
    }
    return false;
}
```

Figure 4-4: The primitive action "spawnZergling" used in JYPOSH. When designing a behaviour class a methods which is referenced by the POSH behaviour as primitive can be accessed by the planner during plan execution without having to explicitly know the underlying behaviour. Using this approach it is possible to have a strong separation between planner and underlying behaviour library. Methods linked to action primitives only return true or false.

directly inserted into the game process. Using python or java within the internal process is more difficult and requires additional changes which would require changes to the JYPOSH system. The EISBOT developed by Weber et al. [2010a] also uses the client-server mode in combination with JNIBWAPI to allow ABL to run independently of the game process.

Taking a closer look at the implementation of the agent, the action primitive given in Figure 4-4 demonstrates the usage of the JYPOSH system. The system allows behaviours to be implemented in either Java or Python; the given action encapsulates internal logic to determine if it is applicable, making it independent of the remaining pool of behaviour primitives. To spawn a Zergling unit within STARCRAFT the action checks if enough resources are available: $mineralCount \geq 50$, $SupplyAvailable \geq 1$ and if the player can create now units. The approach is similar to the precondition checks on plan level. However, those additional checks are intended to guarantee safe execution of a primitive by introducing redundancy.

Once all of the behaviours were implemented and tested, the JNIBWAPI behaviour code was integrated with POSH via Python behaviour modules. These behaviour modules are part of the action selection module in Figure 4-3 and warp the Java methods to be used by POSH to determine which methods are senses or actions. The process of assigning methods to methods in controlled by two lists within a JYPOSH behaviour and needs to be manually added when designing the behaviour class. However, all

of the logic and behaviour for the AI is either in the POSH action plan or the Java behaviour modules. Both can be designed and written independently as long as the planner is able to match primitives to methods. The structure of the behaviour classes was taken from the EISBOT layout [Weber et al., 2010a] as a first starting point.

4.5.3 Iterative Development

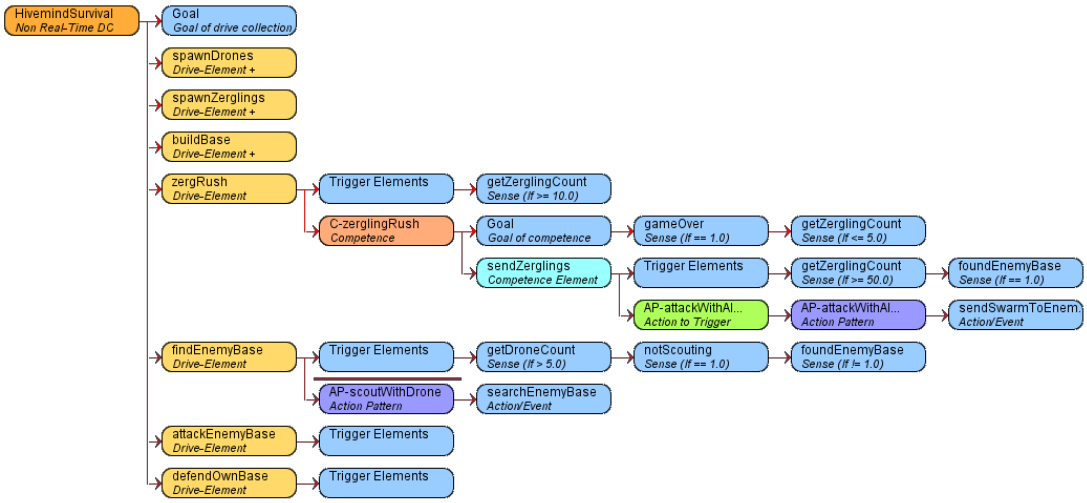


Figure 4-5: Extending the previous plan from Figure 4-2 to include the *Zergling-Rush* strategy. It will send a large group of Units to the enemy base as early as possible in the game.

The BOD architecture provides developers with a structure and a process to quickly develop an AI system provided that the POSH mechanism can be connected to the behaviour modules. BOD's design methodology and process also inform developers how to approach AI problems [Bryson, 2003]. In contrast to the standard waterfall model, development of AI following BOD is an iterative process that focuses on having a flexible set of requirements, and heuristics to re-evaluate the system's decomposition as behaviours and plans are implemented and tested. The plans presented in Figure 4-2 and 4-5 were created using ABODE, introduced and discussed in Chapter 2.3.2, in a visual programming fashion. The editor allows node collapsing which increases the overall readability of the plan because it allows the user to reduce the amount of visible sub-trees present in the tree view. This feature was developed during this project and its need by developers is supported by claims from Chapter 3 with the purpose to aid the understanding of complex behaviours through tool support. The new functionality allows attention to be concentrated on specific parts of the tree while keeping the structure clear.

BOD facilitates the extension of existing behaviours by adding new elements such as being able to use new unit types, thus, complex behaviour can be iteratively developed and step-wise tested. Due to the separation of the behaviour libraries from the POSH plans, the underlying implementation can be adjusted and independently developed creating an interface-like structure. As soon as a behaviour library is developed, different strategies or players can be quickly implemented by shifting priorities within the reactive plan. Furthermore, an advantage of using hierarchical plans is that they are very extensible—new drive elements can be added without any major changes to an existing POSH plan. This easy extensibility is one of the reasons BEHAVIORTREE replaced FSMs as the standard approach to structuring game AI because when extending a FSM a vast amount of state transitions need to be checked (potentially n^2 for n states). Before starting with development the decomposition steps and heuristics which form the development methodology of BOD will be re-iterated, a detailed discussion can be found in Chapter 2.2.9.

BOD Decomposition

1. Identify high-level task of agent
2. Describe activities in sequences of actions; prototype plans
3. Derive action primitives, senses/actions from prototype plans
4. Identify required states; cluster primitives into behaviours
5. Derive goals, drives and order them; prototype drive collection
6. Implement first behaviour from behaviour library

After the decomposition, an initial POSH plan and a first behaviour is coded. The remaining process is the iterative development of writing and enhancing behaviours, enhancing plans and testing them against the environment focusing on rapid development. To guide the iterative changes and process, Bryson and Stein [2001] propose additional heuristics aiding the design following the philosophy: "when in doubt, favor simplicity".

BOD Heuristics

- **Simple structures:** choose a primitive over an action pattern and an action pattern over a competence
- **Reuse:** competence and action patterns can be instantiated multiple times
- **Decompose:** when primitives are too complex
  [decompose into smaller primitives and chain them]
- **Complexity-I:** competences should have three to seven children
  [split competence]
- **Complexity-II:** trigger chains should not be longer than three senses
  [merge into new sense]
- **Execution:** the tree elements should execute quickly
  [use actions only as triggers]

In applying BOD, the first step is to specify at a high-level what the agent is intended to do. To start with the simplest complete strategy—ZERGLING-RUSH—Zerglings have to overwhelm the opponent as early as possible in the game. Based on this more complex strategies can be developed by building on the first one, including re-using its components. The iterative process of starting with a basic strategy is well suited for all complex development processes because the tests and progress of the intended behaviour/outcome can be traced throughout the iterations and unwanted behaviour can be traced back to when it was first included.

To implement a first strategy for the Zerg, several different primitives—actions and senses—have to be coded. The second step after selecting the high-level agenda is to decompose the strategy into sequences of actions needed to build up the strategy which includes building units and sending them somewhere. From there, the required primitives to achieve the sequences need to be identified, e.g. selecting a unit or sensing an opponent unit. For that purpose, an initial top-down analysis is performed to derive the required action sequences for the intended strategy.

Top-Down Analysis:

1. To attack using Zerglings, the AI has to build a mass of those units first and then attack the opponents base.
   ↪ build mass of Zerglings
2. To attack the opponents base, the AI has to know its location.
   ↪ scout the map

3. To build a Zergling, a spawning pool is needed and enough resources have to be available.
   ↪ build spawning pool
   ↪ ensure enough resources

4. To ensure sufficient resources, they have to gather.
   ↪ send drones to gather resources

It is important to remember always to aim for the simplest possible approach first as argued by Bryson [2001]. From the top-down analysis two lists of actions (*mineral_count, support_count, larvae_count, has_spawning_pool, know_opponent_base*) and senses (*harvest_resources, build_spawning_pool, scout_map, build_zerglings, attack_enemy*) can be constructed.

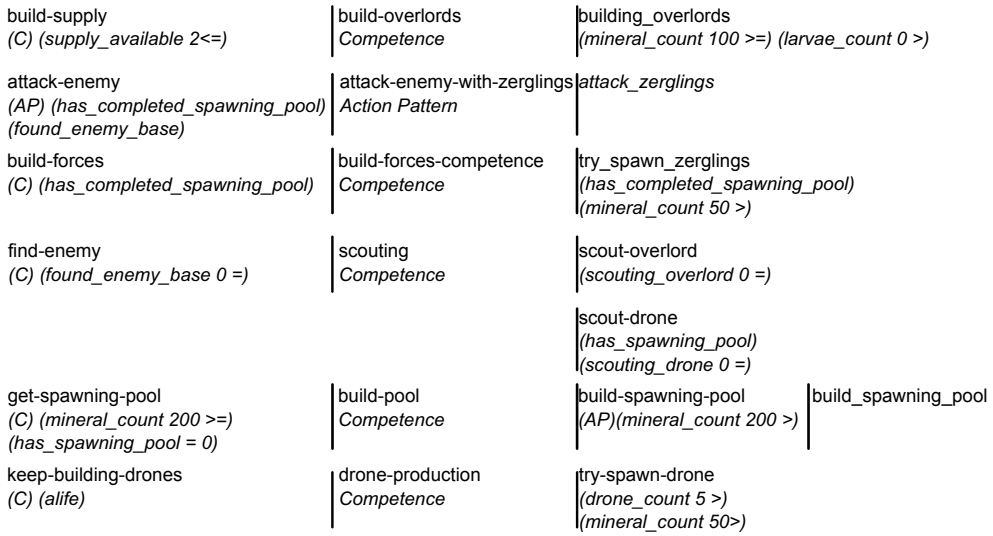


Figure 4-6: A complete POSH plan to attack with a units. The plan extends the previous plan and additionally includes defensive behaviour.

Having derived the first set of behaviour primitives, it is now possible to cluster them in behaviours as is appropriate to their state dependencies. Keeping in mind the basic principles of OOD such as clustering those elements which create classes having a high internal cohesion and a low external one, it is possible to iteratively create a behaviour library. For this case study, it seemed reasonable to create different behaviours for managing structures, building units, combat and exploration & observation. The structure resembles the high-level managers used by Weber et al. [2010a] which indicate the existence of different high-level task groupings within the game and the design need to separate those. Those behaviours for the JYPOSH agent were written in Java, see

the action primitive spawnZergling in Figure 4-4. Once the first behaviours have been developed, a POSH action plan is created to execute them. The plan determines when the AI should call each behaviour to perform an action, in Figure 4-6 an easy-print version of the complete plan is given.

New behaviours were introduced and tested one or two at a time until the plan was robust enough to deal with a full strategy for StarCraft. The iterative process facilitated by Bod allows for building a solid and human-understandable foundation for a first strategy. In Figure 4-5 the extended plan is presented which introduces the *Zerg-Rush* Drive, one of the most prominent early game strategies. The Drive uses the *Zergling-Rush* Competence as a first step. A follow-up step would be the extension of this plan to allow switching between different Rush tactics by including competences for these. These would need to be prioritised differently according to the assessed stage of the game. Mechanisms for varying priorities include code reuse with multiple drive entry points [Partington, 2005] or setting drives to the same priority and having them guarded by mutually exclusive triggers.

Now, a first simple strategy is available which reacts according to the information available in the game and sends in periodic time frames swarms of Zerglings to the enemy base. This strategy works well against human players once or twice before they realise how to counter it. After testing this plan, one will encounter that if the strategy plays against itself, an obvious flaw is present—the absence of a defence mechanism.

Following this process, a very visual and easily traceable design of the bot is introduced which allows the designer more control over the resulting behaviour by increasing the complexity of the aimed behaviour step-wise. The next POSH elements that were developed were those that dealt with the upgrading of units. Then behaviours were added for research, unit production and finally attacking with basic units. The final plan that resulted can be found in Figure 4-6 which creates units when needed, scouts for the opponent, researches unit upgrades and attacks.

As a next step, the underlying primitives can be independently updated without the need to touch the POSH plan. The plan can be improved separately; adding more behaviours for creating different types of units. This is of the strength of Bod over approaches with a stronger coupling of action selection mechanisms and underlying behaviour library such as the agents used with EMAPF [Hagelbäck, 2012].

4.5.4 Results

After finishing the first fully working prototype, the StarCraft agent was tested on the *Azalea*[7] map against the original StarCraft Bots provided by the game, playing adversary races (Protoss, Terran and Zerg) in random mode. The Bod agent performed reasonably well in the first tests winning 32 out of 50 matches, see Table 4.1. The follow-up test against the Artificial Intelligence Using Randomness (AIUR)[8] Protoss bot proved to be a more difficult challenge; the AIUR agent winning 7 out of 10 against Bod. After analysing the performance of the agent, Bod bot competes en par when it is not attacked by a *rush* early in the game, indicating more room for further iterative development for closing the gap between the developed prototype and other available bots.

The major advantage of approaching complex agent design using Bod is the focus on rapid prototyping and continuous working versions of the agent which is incrementally made more sophisticated through continuous testing of the implementation.

Race	Matches count	Wins count	BOD Score AVG	StdDev	Opponent Score AVG	StdDev	Difference Score
Protoss	17	7	19546	35729	43294	18916	-70.75%
Terran	18	12	56651	26077	35696	15380	37.11%
Zerg	15	13	47961	19218	24333	7974	50.64%
Total	50	32	48257	27680	30523	17594	43.56%

Table 4.1: Results from 50 matches of the Behaviour Oriented Design bot presented in Figure 4-6 against the Blizzard AI set to random race on the Azalea Map.

4.5.5 Summarising the Results

Real-time game character development requires either leaving sufficient space for expert human authoring or incorporating large amounts of time and computational resources for automatic adjustments of the agent. This case study focused on the first of those two cases as it allows later adaptation once new requirements emerge due to user feedback.

This underscores the overwhelming importance of systems AI, even where algorithmic AI like machine learning can be exploited to solve regular sub-problems. The clean and visually easy-to-grasp AI produced demonstrates a proof of feasibility. The

[7]The Azalea map is a community-developed map developed for tournaments between professional StarCraft players, see `http://wiki.teamliquid.net/starcraft/Azalea`

[8]AIUR:`http://code.google.com/p/aiurproject/`

usage of a separation of underlying mechanics and behavioural logic allows independent development of both systems. The visual presentation of the plan itself can be a powerful tool because it offers a different perspective on the behaviour. Using features like node collapsing, the plan editor and visualisation tool ABODE also minimises cognitive load when dealing with large plans. The test runs using the original commercial agents of STARCRAFT show good potential—though the developed AI was a prototype representing one of the less-advanced but recognised strategies.

Based on these first results of the prototype, further work on the STARCRAFT AI using BOD seems feasible to allow a closer comparison to more advanced strategies and implementation, e.g. Weber et al. [2010b,a]. First steps would be the inclusion of a more sophisticated strategy such as the *Mutalisk-Rush* and fixing the identified early-game defence problem.

During this development, similarities between BOD and other approaches became apparent. While the BEHAVIORTREE approach introduced by Isla [2005] is widely used in the games industry the highly similar approach POSH has not seen industrial applications. In the Chapter 2.1.1 we are able to discuss BEHAVIORTREE (BT) in more detail and draw more detailed comparisons between the most dominant approach in game AI in Chapter 2.4. Similar findings are present in the results of the analysis of different design approaches in Chapter 3 where the need to enhance tool support is identified,.e.g. enhance ABODE to provide real-time debugging, offering the AI designer useful feedback and statistics on the developed plan. Other directions which are clearly visible are the inclusion of lower-level approaches such as potential fields, neural networks or MCTS in the context of BOD behaviours which might benefit from providing the high-level POSH control with a greater level of flexibility. However, the more external approaches are involved, the more complex the resulting agent systems gets and thus, the more tool support and process structure is needed to guide the developer.

4.6 Advanced Planning for StarCraft

Within the first case study, a prototype able to beat the commercial AI in 64% of the matches has been presented. The developed strategy followed a top-down analysis of implementing a *Zerg Rush*, see Figure 4-6. Several arguments such as the existing weakness of the presented approach and the low sophistication leave room for improvement. In this section an extension of the presented argument for supporting game AI is discussed aiming towards DEEPER AGENT BEHAVIOUR.

```
1 9 - Spawning Pool
2 8 - Drone
3 9 - Extractor
4 8 - Overlord
5 8 - Drone
6 @100% Extractor - send 3 Drones to mine gas
7 @100% Spawning Pool - 6 Zerglings
8 @88 Gas - take Drones off of gas one by one, giving you 104 gas & Research
      Speed.
9 Pump Zerglings, after first Overlord get more Zerglings.
```

Figure 4-7: An encoded strategy for StarCraft. The strategy is called "ThreeHatchZergling-9Pool" and is intended as an early game strategy to throw off the opponent. More details on the strategy are available at: `http://wiki.teamliquid.net/starcraft/3_Hatch_Zergling_(vs._Protoss)`

4.6.1 Encoding User Knowledge

By using encoded strategies from StarCraft user forums and offering a means to translate those in a way the players are familiar with, an approach for designing a modular real-time strategy AI for StarCraft that is more novice friendly can be derived. In the user forums such as Liquidpedia[9], players discuss match strategies using a loose protocol which describes decision points and strategies. Such an example notation is given in Figure 4-7.

The protocol can be interpreted as production rules which are organised in a stack-like order. In Figure 4-7 a production is given by `9 - Spawning Pool`. In this production, the first number defines the amount of `supply` that is used by the player followed by the unit to build. `supply` is an implicit resource which determines the number of game units the player is able to control. `supply` in contrast to the other two resources, *Gas* and *Chrystal*, can not be gathered or mined but is gained when upgrading buildings or specific units. After the amount of supply is given the task is assigned, in this case, to build a `Spawning Pool`. Once a rule has been applied, it is moved off the stack, executing the next rule. The second type of rule is based on goal completion identified with an `@`, such as gathering a certain amount of resources in @88*Gas*, on line 8 or that research of a building reached a certain completion rate, e.g. `@100% Extractor`.

Such high-level strategies can also be encoded as a nested condition block using `if -else` rules. Nonetheless, the usage of a more flexible system such as a reactive planner allows for better responses to dynamic changes in the environment such as the strategy being not applicable anymore or how to respond the when a currently active goal

[9]Liquidpedia is an online community centred around RTS games, discussions about employed strategies and professional competitions, see `http://wiki.teamliquid.net/starcraft`

cannot be pursued anymore. Additionally, the control and task assignment of when and how to attack the opponent can rarely be contained in those simple rules. It is hard to formalise productions in rules in sufficient detail while maintaining a small rules collection. In most cases, the users specify additional properties such as when to attack as verbose text blocks due to a missing way of expressing them in another form. A possible hypothesis is that Behaviour-Oriented Design can provide such an expressive form.

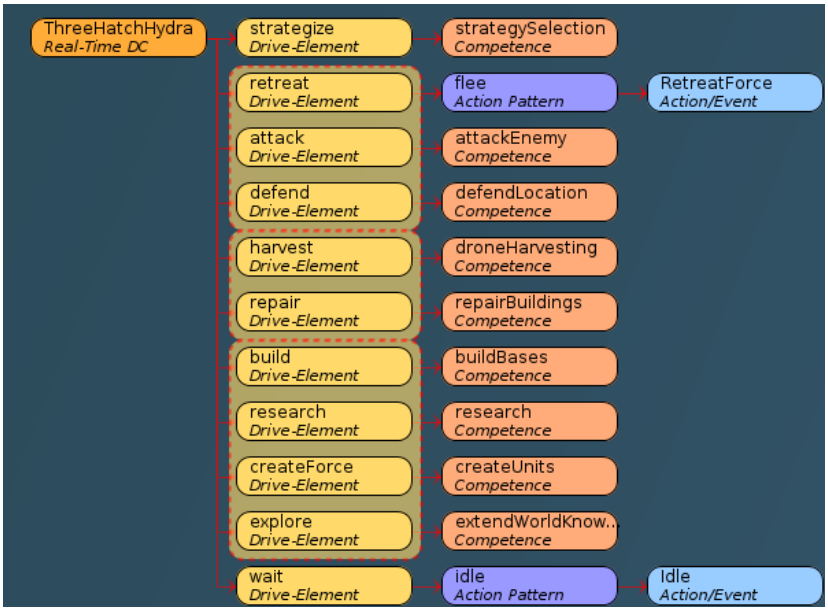


Figure 4-8: The *ThreeHatchHydra* POSH plan for the extended StarCraft agent. The presented plan only visualises the 11 Drives which form the agent and their contained competences. The full plan is available in the appendix B-1. The drive elements are clustered based on their priority. However, drives of equal priority form groups which are surrounded by dotted lines, e.g. harvest and repair.

4.6.2 Extending beyond individual strategies

In the case study the development, implementation and evaluation of the Bod agent was driven by the top-down analysis of the *Zergling rush*. An altogether different starting-point would have been the usage of the protocol to derive the needed primitives and high-level goals used by the planner.

Based on the previous hypothesis that an approach similar to the forum users' protocol can be designed, a new implementation of the Bod agent for StarCraft is presented. Taking into account the System-Specific Step requirement from Chapter 3 and integrating the analysis from Chapter 2.4, a new action selection mechanism,

```
9  - Overlord
9  - 3 Drones
12 - Hatchery (at natural)
11 - Spawning pool
10 - 4 Drones
14 - Hatchery
@100% Spawning pool - 4 Zerglings
15 Extractor
16 - Overlord
@50 Gas - Hydralisk Den
@100% Hydralisk Den - Hydralisk speed upgrade, 2nd extractor
26 - start massing Hydralisks
@100% Hydralisk speed upgrade - Hydralisk range upgrade, take 2 Drones of
    2nd Extractor
@90% Hydralisk Range - Move out
```

Figure 4-9: An encoded strategy for StarCraft. The strategy is called "ThreeHatchHydra" and is intended for a Zerg-Protoss pairing. More details on the strategy are available at: `http://wiki.teamliquid.net/starcraft/3_Hatch_Hydralisk_(vs._Protoss)`

posh-sharp, is introduced in Chapter 5. This new mechanism allows the implementation of agents in a more directed way utilising Bod.

As a second prototype, a more sophisticated strategy has been selected—the *Three Hatch Hydra*. This strategy performs well in mid-games and is on par with strategies often occurring at competitions. Using as a starting-point the protocol given in Figure 4-9, a top-down approach can be described in much more detail and well guided while remaining true to the original recorded strategy. Bod in combination with Abode allows for a nearly one-to-one translation of the protocol into a partial plan. When integrating the conceptual ideas of manager into the plan, it is possible to derive posh *Drives* which resemble manager responsibility. Yet, in contrast to the manager concept used by Weber et al. [2011], there is no need to cluster the underlying behaviour layer in a way which does not resemble their functional similarity. As an example, the building manager might need access to units but it should not create linkage to the other manager because it would render the modules highly cohesive.

The new posh plan, see Figure 4-8, contains the user plan given in Figure 4-9 as a competence which is used by the Drive for building a base. This structure allows the agent to arbitrate between different "desired" Drives resulting in a more flexible agent.

The "build" competence for the *ThreeHatchHydra* plan from Figure 4-9 is implemented in Abode in Figure 4-10. This competence is the derivative of integrating the full protocol as is and contains nine elements responsible for triggering the correct sub-tree at the right time. The present behaviour goes against the Bod metrics and

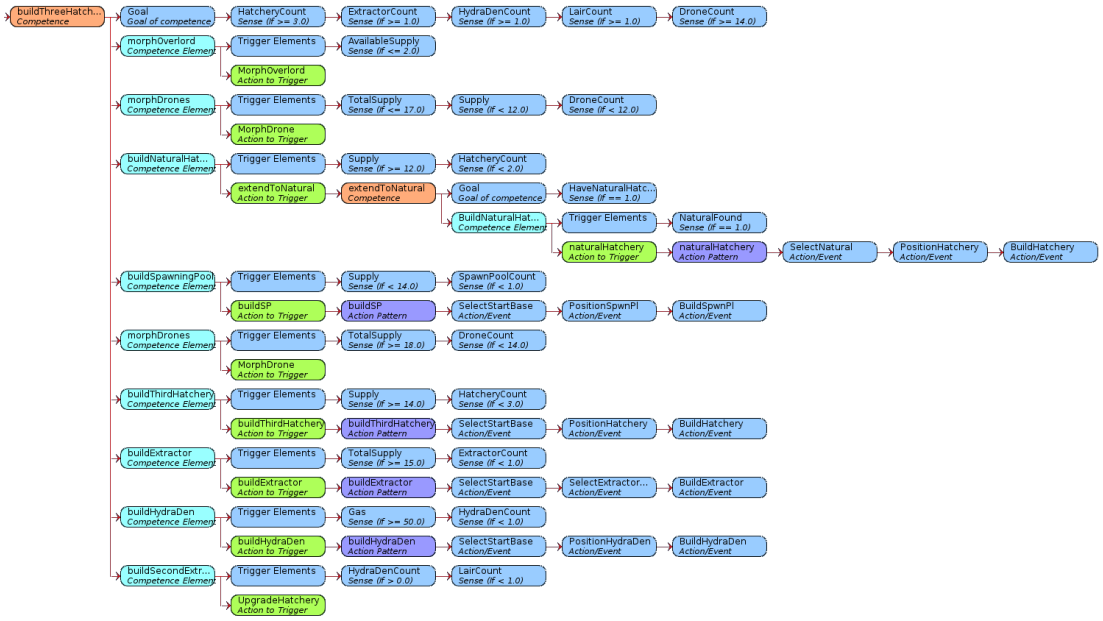


Figure 4-10: The full build Competence of the *ThreeHatchHydra* strategy which by design following the top-down analysis resembles Figure 4-9.

indicates the need for refactoring it into smaller competences, however, the hypothesis of being able to represent plans in a format familiar to the forum users illustrates the flexibility of the presented approach. Once, a sufficiently large behaviour library for POSH exists it is possible to re-build most of the strategies from the forum without the need for touching the underlying code.

Additionally, the separation of logic and implementation has been strengthened by utilising a new mechanism in POSH-SHARP for inspecting behaviours and making their actions and senses accessible, as shown in Figure 4-11.

Figure 4-11 demonstrates one of the action primitives which can be used by POSH-SHARP and the planner to select a location on the game map for a spawning pool. The action also illustrates the strong separation paradigm removing the need to specify action primitives by hand, the new mechanism will be introduced in Chapter 5. Each behaviour class is independent of other behaviours to allow easier transitions between different implementation. A newly introduced idiom in POSH-SHARP is the *Behaviour Bridge*, the Interface (). Instead of using a global blackboard as done in other approaches such as Goal-Oriented Planning (GOAP) [Orkin, 2005], each behaviour has an independent short-term memory but the *Behaviour Bridge* shares access to the most desired elements by providing a central access point. In contrast to other POSH behaviours, the Interface () as the name suggests "hides" its implementation from the

```
        /// <summary>
        /// Select suitable location for the spawning pool
        /// </summary>
        /// <returns></returns>
        [ExecutableAction("PositionSpwnPl",1.0f)]
        public bool PositionSpwnPl()
        {
            if (!Interface().baseLocations.ContainsKey((int)Interface().
                currentBuildSite))
                return false;
            // TODO: this needs to be changed to a better location around
                the base taking exits and resources into account
            TilePosition buildPosition = Interface().baseLocations[(int)
                Interface().currentBuildSite];
            builder = Interface().GetBuilder(buildPosition);

            buildPosition = PossibleBuildLocation(buildPosition, 1, 1,
                100, builder, bwapi.UnitTypes_Zerg_Spawning_Pool);
            buildLocation = buildPosition;

            if (buildPosition is TilePosition)
            {
                move(new Position(buildPosition), builder);
                return true;
            }
            return false;
        }
```

Figure 4-11: A behaviour primitive of the BuildingControl behaviour. The action is responsible for selecting the correct location of a SpawningPool . The POSH-SHARP annotation [ExecutableAction("PositionSpwnPl",1.0f)] allows a more robust configuration of actions and senses than JYPOSH by allowing the plan to link to different methods and use the name provided by the annotation to link to a given action or sense.

other classes and is shared at instantiation time of the agent with all POSH behaviours that are generated for the agent. Thereby, the resulting agent behaviours can be modified independently allowing each behaviour to communicate throughout the agent with other behaviours loosely. The *Behaviour Bridge* itself does not actively participate in the agent's behaviour expressions, however, it allows a double-blind access to crucial perceptual information.

4.7 Concluding Real-Time Strategy AI Contributions

In this chapter, a cognitive approach—Behaviour-Oriented Design—to Real-Time Strategy games was discussed and demonstrated in a case study. Based on the results, the Bod methodology offers sufficient capabilities to handle complex, highly-demanding environments while still being a light-weight cognitive planner. The approach from the case study was compared to a classical baseline approach—the commercial AI contained in the game. The designed agent in the case study was able to win against the commercial AI in over 60% of the matches which indicate a promising approach to investigate further.

Approaches such as EisBot by Weber et al. [2011] utilise a more powerful planning mechanism to control the agent. They put much work into the underlying behaviour library and the manager classes. However, adjusting the strategy to shift towards different known strategies is hard as all the behaviours are hand-coded and the Abl planner creates the resulting Abl tree based on the current goals. Other approaches such as Evolutionary Potential Fields are promising but they are unable to handle large scale coordination and strategy design. Similar to neural networks, EMAPFs are used as black box solutions which restrict the design space.

After discussing the case study, a new approach using human encoded strategies was discussed encoding strategies from StarCraft user forums into sophisticated agents. The approach offers a robust and flexible interface for altering the strategy even by novice users based on the similarity to the original encoding. The resulting agent is more complex and contains a human competitive strategy. The advanced planning approach demonstrates capabilities that go beyond simple plans by developing large plans for game agents. This approach demonstrates a first application of human knowledge in the form of strategic build plans that allow even novice users to develop advanced agents using a visual editing of posh plans.

In the next chapter, a new design methodology—Agile Behaviour Design is presented which extends its predecessor Bod to accommodate the requirements for developing complex game agents in an industrial environment. In addition to the methodology, the chapter also includes a new agent framework accompanying the methodology for robust agent development in highly restrictive environments such as the web or smartphones.

Chapter 5

Advancing Tool Supported Action Selection

In the previous chapter, two approaches were presented for developing agents in the highly demanding domain of Real-Time Strategy (rts) games. Whereas the first demonstrates the general application of Behaviour-Oriented Design (Bod), the second approach demonstrates a highly complex agent, integrating human encoded expert knowledge into the core of the game AI logic. Due to the loose coupling of the plan and implementation by having separate representations and the strong resemblance of the agent plan and the original strategy, it is possible to enable novice developers to create sophisticated agents, even when not being able to program.

In this chapter, a new methodology for agent development is proposed extending the existing approach Bod. This new process is designed to support work habits in industrial environments and multi-disciplinary teams. To aid the new methodology and to address the identified issues in current approaches from Chapter 3 such as missing debug support, complex set-ups and implicit design rules, a new agent framework—posh-sharp—is proposed. This new framework was implemented in C# and contains sample code for different game environments. It is available online under: `https://github.com/suegy/posh-sharp`. The framework allows the development of agents for highly restrictive environments such as mobile phones or web browsers and includes advanced features from current software development.

5.1 Contribution

This chapter contains a proposal for creating a new real-time game AI approach that extends and alters an existing, established methodology for creating behaviour-based

AI—Bod. The proposal draws on the System-Specific Step (sss) and the analysis of StarCraft bot design based on the Eisbot by Weber et al. [2011], the EMAPF bot from Hagelbäck [2012] and the two agents described in Chapter 4. To complement the new approach, I developed the posh-sharp framework including a new version of Parallel-Rooted Ordered Slip-Stack Hierarchical (POSH) action selection which addresses issues in the existing Bod architecture of posh action selection and focuses on easy integration and maintainability.

In Chapter 3 the sss is derived by analysing existing architectures and their usage by experts. A sophisticated agent for a rts games and its implementation is elaborated in Chapter 4.3 which requires planning and strategising on multiple levels of abstraction. Based on the findings during both processes it is possible to present advancements which address some of the core platform issues identified in the system in this chapter.This chapter does not include published content and my contribution to it is 100%.

5.2 Agile Behaviour Design for Games

Chapter 1.2.1 presents a scrum variant for games introduced by Keith [2010]. This variant of scrum aims to support the development process of creative products such as games better compared to traditional software development processes. The process itself is agile by design and allows alterations and new features to be introduced late in the project. "scrum for games" contains four phases (concept, pre-production, production, post-production) which have defined milestone points that are generated at the start of the project but can be shifted slightly based on the project's needs.

The Bod methodology describes an iterative process working in similar ways to agile processes such as scrum but the steps to arrive at the desired outcome are not aimed at developing larger AI systems containing a vast amount of Competences and behaviours focusing on *in-time* development. Thus, to make the process more controllable, integrating design steps from Keith [2010] can enhance the process while making it similar to the one already used in game development which allows for an easier transition from the currently used method.

The phases of game development impact the creative freedom of a designer and influence the system design, the more mature the system becomes, the more restrictive it becomes in terms of possible deviations from the initial design. Thus, features need to be known as early as possible and should not emerge continuously due to iterative changes to the system.

Game Development

- **Pre-Production:** This phase combines the concept phase (which is sometimes a separate phase) and the development of early prototypes which are presented to a producer. The concept and the prototype offer a great freedom but do not have to reflect the final product. This phase focuses on strong design and the architecture and approach for the game are selected.

- **Production:** During this phase, the game design is still altered and work on the game commences, a step which could also involve adjusting or developing a game engine. This phase has three internal milestones: Alpha, Beta and Gold, which reflect the maturity of the game. As part of this phase, the design document from the accepted concept gets realised and the more the game matures, the more rigid the design deviations become. However, after each reaching Alpha phase, the game will be tested and based on the results of the testing major re-work is required. The same applies to the Beta milestone. However, changes to the underlying system are harder to realise, thus, the game design itself gets adapted. A more flexible system would allow for more freedom in terms of design.

- **Post-Production:** During this phase, only minor changes are made to the game to address bugs. However, if additional releases such as Downloadable Contents (dlcs) are planned, the system can be made more flexible again to integrate new features. This opening up of the system increases the design freedom again.

The original approach to Bod is a top-down analysis of a desired behaviour combined with a bottom-up generation of plans and the behaviour library. The top-down analysis starts with the definition of a high-level task the agent wants to achieve, an undertaking possible for generating a single agent in a well-defined environment by an expert. During the analysis of the sss and the introduction of the approach to developers from other approaches, in this case Abl, the decomposition and plan generation was found to be a challenging process. Similar observations were also made during the *Intelligent Control and Cognitive Systems* course where student as part of their coursework on creating Interactive Virtual Agents (IVAs) are using Bod. Novice users tend to generate either flat shallow plans or deep narrow plans, restricting the resulting agents immensely. This also applies to other approaches such as BehaviorTree (Bt).

When using Bod, for early iterations, iterating over the plan and creating new behaviour primitives does not result in the desired decoupling of programmer and designer as the complete behaviour library and plan structure is in flux.

BOD—Initial Decomposition

1. Identify and clearly state the high-level task of the agent.
2. Describe activities which the agent should be able to perform in terms of sequences of actions. Prototype first reactive plans.
3. Derive perceptual and action primitives from those initial plans and sequences.
4. Identify required states and cluster primitives based on shared states into behaviours.
5. Derive goals and drives and order them according to their intended priorities. Prototype drive collections using those drives.
6. Implementation of a first behaviour from the behaviour library.

The original BOD decomposition based on Bryson [2001] results in a minimal plan that is often not sufficing the original intended goal of the agent but a boiled down version only showing a proof of concept implementation [Partington and Bryson, 2005]. In their application, Partington and Bryson [2005] described the initial task of an agent to capture the enemy flag from the opponent base. After the decomposition, as described above, a list of action sequences exist and a plan which contained only a small number of drives and actions. The resulting agent is able to perform a basic task such a moving in a direction but not sufficing the original goal.

From this initial prototype, the plan incrementally turns more complex by adding new elements. While increasing the complexity of the plan, the first decomposition and primitives list get adjusted as unforeseen options now need to be considered. However, this process requires revisiting the underlying behaviour library, a process which creates strong coupling between designer and programmer. Additionally, the process requires reconsidering the intended high-level task leading many times to re-interpreting the existing plan each time.

Instead, it is possible to start the process with a different intention by taking the scrum process into account when designing an agent. Scrum is an agile software development process integrating iterative development and testing while maintaining as much as possible the time predictability from other development processes such as the Waterfall model. Scrum partitions the project into smaller *Sprints* that take a fixed time and each *Sprint* deals with a defined set of features. At the end of a *Sprint*, the entire system is supposed to be able to execute the features developed during the Sprint, including those that have been newly integrated. The features are collected on a feature board which presents all features in ordered lists (product backlog, *Sprint*

selection, in-progress, in-review, completed) of completed, in-progress and to be implemented elements. Scrum starts with an initial full specification of the system and continuous stable versions of the product while incrementally adding features from a feature board. The important part is the feature board; it is created and laid out to schedule the work and progress of all features. The work on the product starts after all features for the final product have been laid out. After a feature has been integrated, the full system needs to be in a stable state again. The current BOD is similar but as shown by Partington and Bryson [2005], the starting point is a minimal plan and thus a minimal set of action primitives.

Agile Process Steps

- Due to the modularity of POSH the scrum approach suits BOD perfectly. After the initial decomposition, instead of creating a small prototype plan and start working on a first behaviour we can take a different route.

- The first goal is to have a full plan for the agent containing all intended drives and primitives according to the specification. This step is time-consuming and cognitive challenging. The resulting plan should suffice the agent task. This step should be done in one go and should not stop at a high-level description in terms of drives and a small set of Competences but instead, should include intended sensory and action primitives from the decomposition. This part of the development is a pure design stage without the need for programmer involvement.

- The initial design is evaluated with a programmer, taking into consideration actions and sense primitives. As a guideline actions and senses should describe activities such as moveToTarget or seeEnemy. The initial design is adjusted based on programmer feedback to integrate and adjust naming of underlying primitives.

- Behaviour stubs are generated in a rough class structure, creating hollow primitives in POSH behaviours for all actions and senses contained in the plan. This stage is hidden from the designer and is a pure programmer task which is easy to implement.

- All empty action primitives should contain a default return state representing failing actions. The only action which needs to be implemented at this point is the fallback action which is controlled by the lowest priority drive in the plan. This action should represent an idle state requiring nearly no resource or complex implementation.

- When designing the plan it is important to keep in mind that senses are triggered upon meeting a condition. Thus, when the conditions are not fulfilled for a single sense the trigger does not release the related drive or competence as all senses within a trigger are joined by a logical and. Using this knowledge, it is possible to deactivate parts of the POSH tree similar to the bitmasks used by Isla [2005] see Chapter 2.1.1. To achieve this, the designer integrates as the first sense in the Trigger a success sense which on the underlying behaviour side only returns true. As a condition check, this sense compares against false which leads the sense always to fail. The condition check needs to be specified on the designer's side. Once a sub-tree has been implemented the success trigger is removed.

- After obtaining a first feature-complete plan, the work on the underlying behaviour primitives can be adjusted to work on individual features. Thus, the feature board can be ordered by clustering actions and senses under specific features. The alteration to the feature board can be done by grouping actions and senses according to their position in the hierarchical tree. A feature on the board then relates to all actions and senses contained in a specific competence. This supports the identification of redundant Competences or functionality that can be reused by identifying similar usage of actions and senses within competences.

- On the feature board, the relating features should be ordered according to their correlating drives to focus on completing drives one at a time. This clustering of features allows programmers to shift entire feature blocks up and down on the feature board without impacting other sub-trees. As an example, let us look at the competence in Figure 4-10 on page 162.

- If the behaviour designer decides to alter the plan at this point, a large number of actions and senses are already stubbed within the hollow behaviour set. This given structure allows the designer to work independently on the design while programmers can implement the stubs, one at a time. Following this approach requires fewer inclusions of new underlying primitives than following a simple incremental approach; it also distributes the work better between designer and programmer by initially close coordination in the first phase and a looser coupling later on.

- Once a new feature is integrated, the sub-trees can be activated by removing the locking sense. This removal automatically unlocks more and more of the designed agent behaviour. Ideally, the work is directed from bottom to top through the POSH plan following the idea of the SUBSUMPTION design of Brooks [1986]. This

will enable higher level drives after lower level ones have been implemented and tested increasing the complexity of the agent according to the designed priorities.

This altered approach was used to develop the StarCraft agent presented in Chapter 4.6 and is utilised in the Android game Stealthier POSH[1]. Maintaining a prioritised feature set which relates to the sub-trees (Competences or Drives) proved in those two early case studies beneficial and reflects the work on commercial games better due to the usage of a feature board. The board allows to track the development progress and allows for more independent work of designer and programmer. Additionally, it removes the burden of numerous changes to the behaviour library early in the project or restricting the designer from working purely on the plan without being able to test it. Unimplemented actions return the default false state so even partial competence respond adequately.

5.2.1 Handling Complexity

With increased size of an agent, the complexity of the underlying behaviour library and the plan structure grows as well. If the system is based on FSMs, the complexity would, in the average case, increase exponentially which would render any system at a certain complexity unusable for human editing. This is based on the assumption that the high connectivity of states and the required checking of state transitions between them reduces the understanding of the system and the ability to maintain it. With approaches such as Bt and posh, the complexity grows only in the worst case exponentially, in the average case, the complexity increase is lower. To manage this growth, several support mechanisms were identified. One mechanism is planning systems such as Goal-Oriented Planning (Goap) [Orkin, 2005], they require expert knowledge of the plan to predict the outcome but reduce the interdependence of nodes and the amount of manual checking transitions. posh as a lightweight planner allows local design by modifying existing Competences due to the ability to nest Competences and the hierarchical structure of the drive collection. As Competences are re-used and handled by the planner, the amount of connections which need to be adjusted is similarly low compared to other reactive planners. In combination with the proposed agile Bod it is possible to work on smaller sections of an agent by focusing on Drives and Competences while the dependencies between designer and programmer are reduced. Using the previous approach of default trigger states, sub-trees unlock based on the progress of their underlying implementation. This cascaded unlocking of the tree and the resulting behaviour allows for a better version control of the behaviour library because it

[1]The game is available on the Android app store or using the following link: `https://play.google.com/store/apps/details?id=com.fairrats.POSH`

is more directed towards realising connected sub-trees. Additionally, the current POSH editor Advanced Behaviour-Oriented Design Editor (ABODE) supports this process by collapsing currently not interesting tree nodes; a feature also present in most of the BT editors presented in Chapter 2.3. The combination of working on sub-trees and the feature board based on scrum directs the agent implementation to focus on connected pieces. In combination with the POSH-SHARP reactive planner and the therefore reduced dependencies between tree nodes, the new approach should provide sufficient support for working on more complex systems.

To go beyond the support of JYPOSH POSH-SHARP is providing additional software design support.

5.3 POSH-SHARP

To enhance the support of game AI development, a new arbitration architecture is proposed which alters the structure of the existing JYPOSH system and contains four major enhancements: multi-platform integration, behaviour inspection, behaviour versioning and the *Behaviour Bridge*.

The new system switches the implementation language from Java&Python to Microsoft's C#—a platform-independent language which in contrast to Oracle's Java is fully open-source. Additionally, a resulting agent can be better integrated into most commercial products based on the usage of a new deployment model of the system—the dynamic libraries (DLL). The POSH-SHARP DLLs allow a developer to integrate the POSH behaviour arbitration system into any system which supports external libraries. The strength of this method in contrast to JYPOSH is the removal of the dependency on a JAVA virtual machine or a Python installation. This reduces the configuration time and potential problems with incompatibilities or wrong setups. POSH-SHARP was designed to work on computationally less powerful devices such as smartphones or in the web-browser emphasising the lightweight nature of POSH. To guarantee this POSH-SHARP is mono 2.0 compliant[2]. The POSH-SHARP architecture is separated into different distinct modules to allow the developer, similar to the node collapsing in plans, to focus on smaller pieces of source-code and fewer files. The previous JYPOSH system required a complex setup and required the developer to maintain a complex folder structure which contained all sources for POSH and the behaviour library. To support and extend the separation of logic and implementation most languages use some form of container format. In JAVA modules are clustered and distributed in *Jar*

[2]The Mono project provides a free C# platform-independent library supported by Microsoft. Mono 2.0 is the language level used for mobile devices and in the Unity game engine is used for full cross-platform compatibility. Mono is available at: `http://www.mono-project.com`

files and in Python *egg* files. This helps reduce the burden of a programmer to maintain a manageable code base.

5.3.1 POSH-SHARP Modules

- The **launcher** is the smallest module. It is responsible for selecting which plan to load, to tell the planner to construct a POSH tree based on a serialised plan and finally to connect the **core** to the environment. The launcher receives upon start a set of parameters containing an agent definition and link to the environment. The launcher then calls the core and specifies which agent is connected to which plan. It additionally makes the behaviour library in the form of dlls accessible to the core. The launcher is platform dependent and is available for Mac and Windows and can be re-compiled based on the project's needs. For the Unity game engine[3] a specific launcher exists and integrates fully into the game engine.

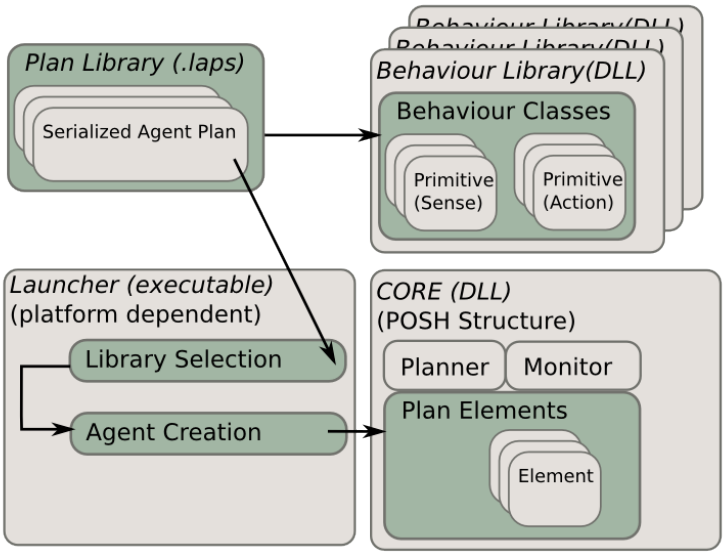


Figure 5-1: The POSH-SHARP module structure. The system consists of the core dll, the launcher executable, a set of plan files and behaviour library dlls. A minimal system contains four files and does not require a special setup on a host machine.

- The **core** module is platform independent and can be used "as-is" as it does not rely on other modules, see Figure 5-1. As a first step, the core instantiates a POSH agent responsible for enveloping the POSH tree and the connected behaviour objects with their contained short-term memory. After creating an agent shell,

[3]Unity is a fully featured commercial game engine which supports the cross-platform development of games and is available at: `http://unity3d.com/`

the planner uses the serialised plan file to instantiate a POSH tree for the agent. For that, it inspects the behaviour libraries and instantiates all behaviours for the agent which contain primitives required by the serialised plan. This process is done for each agent. After all agents embed a live POSH tree, the core links the agent to the environment exposing the sensory primitives to receive information and the action primitives to interact with it. The core also contains a monitor for each agent that allows live debugging and tracing of agent behaviour.

- A **behaviour library** is a self-contained set of behaviour classes wrapped in a dynamic library file (DLL). They are coded by a programmer and implement the functionality used in conjunction with a POSH plan. The behaviour classes contain POSH action and senses, as illustrated in Figure 5-2. The advantage over JYPOSH is that the core automatically inspects all behaviours and loads only those who are correctly annotated. Thus, there is no need to specify a list of actions and senses within the header of a behaviour. Additionally, behaviour primitives can be "versioned", a new feature in POSH-SHARP which offers the programmer a way to develop an agent incrementally without overriding and deleting working functionality.

- The last component of POSH is the plan library which contains a collection of POSH plans. The POSH-SHARP plans are identical to the JYPOSH plans allowing users to migrate their plans to different systems. The plans are in a Lisp-like syntax and can be interpreted as serialised POSH trees that are used by the planner.

5.3.2 Behaviour Inspection & Primitive Versioning

In previous versions of POSH, behaviours had to contain lists of string names referencing behaviour primitives to be used upon loading the class. Additionally, all behaviours had to be in a behaviour library folder in source format. This behaviour folder was inside the same folder hierarchy as the POSH system, also as source files. This project structure forces developers to maintain and manage more files than necessary, it reduces the visibility of own behaviours and increases the chance of modifying or removing essential parts of POSH unwillingly. POSH-SHARP introduces the packaged POSH *core*, combining the planner and the entire structure of the system into an 111kB sized dynamic library file. Behaviour files are also compiled into behaviour library DLLs which is supported by free tools such as Xamarin's Monodevelop[4]. Upon starting POSH-SHARP, the core receives as a parameter a list of dynamic libraries which should be inspected.

[4]Monodevelop is an open-source Mono/C# IDE available at: `http://www.monodevelop.com/`

Once the POSH plan is loaded, POSH-SHARP inspects all libraries and loads all that contain annotated primitives which are referred to by the currently active serialised plan. Using dynamic libraries reduces the number of files developers and users have to handle and reduces the risk of erroneous handling of files.

```
[ExecutableAction("a_charge", 0.01f)]
public void Recharging()
{
    // Set an appropriate speed for the NavMeshAgent.
    Loom.QueueOnMainThread(() =>
    {
        if (nav.speed != patrolSpeed)
            nav.speed = patrolSpeed;

        // Set the destination to the charging WayPoint.
        navDestination = charging.chargerLocation.position;

        if (nav.destination != navDestination)
        {
            nav.destination = navDestination;
            nav.Resume();
        }
        // If near the next waypoint or there is no destination...
        if (nav.remainingDistance < nav.stoppingDistance && nextToCharger)
        {
            nav.Stop();
            //asynchron charge batteries
            Loom.RunAsync(() =>
            {
                charging.Charging();
            });
        }
    });
}
```

Figure 5-2: A behaviour primitive for recharging a robot within the STEALTHIER POSH Android game. The action uses a NavMesh to determine the position of the agent and then charger the robot once the agent is close enough to the charger. To allow for threading a scheduler (Loom) is used to outsource specific tasks into Unity's internal update thread. The action is set to version 0.01 which allows later actions to override the behaviour and the action links to the plan name a_chargeMore details on the game are available at: `https://play.google.com/store/apps/details?id=com.fairrats.POSH`

The behaviour inspection uses the specific POSH annotations to identify primitives within a behaviour library file. There are two standards annotation classes ExecutableAction and ExecutableSense, both augment a method and attach a name referenced and search for by the player and a version number. In Figure 5-2 an example action from the STEALTHIER POSH Android game is given. The primitive is called by

the planner when the robot agent needs to recharge the battery and uses a NavMesh[5] to identify if the agent is spatially close to a charger. As described in Chapter 2.2.9, primitives should be as independent as possible and use their perception. In this case checking the internal state of the NavMesh. By removing the need for specifying the string name of a method in a specific list, a potential risk of mistakes is removed from the development process. The usage of the extra name tag allows the usage of names which would otherwise break the naming convention of C# and allows for more descriptive and customised names.

The behaviour primitive versioning uses the second parameter of the annotation. The planner in default mode always selects at run-time the primitive with the highest version number. This mechanism allows the planner to exchange primitives during execution if needed. Dynamic primitive switching is a complex process and needs further investigation and feedback from the user community. However, the overloading of existing primitives at design-time is a powerful process which allows developers to extend functionality by following the idea of Brook's SUBSUMPTION idiom.

5.3.3 Memory & Encapsulation

Similar to architectures such as ACT-R and Soar, POSH-SHARP provides a centralised way to store and access perceptual information about the environment. Game environments have strong restrictions on computation, thus, polling sensors which require computation or perform continuous checks should be as rarely used as possible. The usage of a fair amount of polling sensors reduces the time the agent has to undertake the actual reasoning. The *Behaviour Bridge* illustrated in Figure 5-3 provides a centralised access to perceptual information acquired from the game environment. Each individual behaviour is able to access and share this information and use it internally. In a sense the *Behaviour Bridge* is to some degree similar in its function to the *corpus callosum* in the mammalian brain. It offers an interface between parts which are spatially separated due to their distance in the brain and provides a fast and efficient means of information exchange. It is designed around the software *Listener Pattern*, making game information available to all subscribed behaviours. When removed or damaged most of the brain still functions, however, some functions are then erroneous or slower. The same applies to the *Behaviour Bridge* as it allows information exchange but does not undertake actual communication or computation.

Memory, same as in other POSH versions, is contained within individual behaviours. There is a strong argument for self-contained behaviours and their internal memory

[5]NavMeshs have been discussed in Chapter 2.1.2 as a way of structuring virtual environments more efficiently.

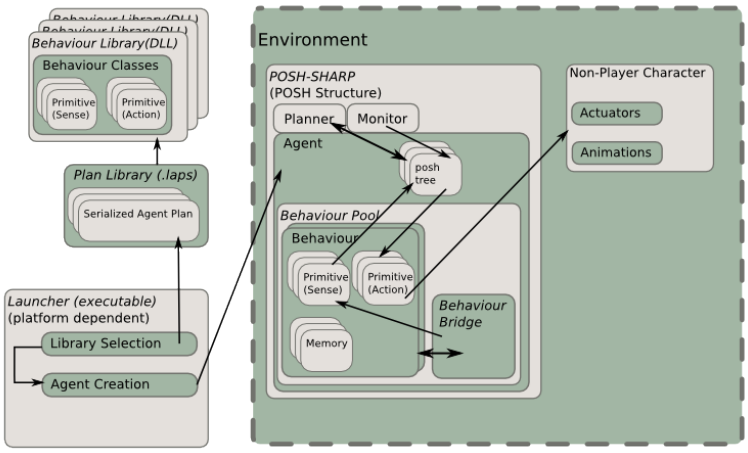


Figure 5-3: The POSH-SHARP architecture once the modules have been integrated into an environment, e.g. the integration with a game engine such as Unity.

which is, that their usage supports lower inter-dependencies between behaviours and fosters the modularisation & exchange of behaviours. POSH-SHARP supports this exchange through behaviour library files which offer easy exchange by swapping out individual dynamic library files. Thus, a general focus on a specific class in a library outside the *core* could break the entire agent.

A global blackboard as part of the architecture is currently not supported by POSH-SHARP, even though the integration would be easy using the *Behaviour Bridge*. The usage of a blackboard or long-term memory, similar to the memory types by Orkin [2005] or the *Working Memory Elements* of ABL, introduces extra complexity into the design process which is may not be desirable for a light-weight novice-oriented architecture. Behaviour designers using a blackboard need to take potential memory into account when designing behaviours. This means that the memory emerges and changes over the course of the experience, requiring additional careful design and anticipation of behaviours interacting with it.

Instead of a global blackboard which offers reading and writing complex information from it, POSH-SHARPprovides the *Behaviour Bridge*. Using the *Behaviour Bridge*, POSH-SHARP provides a centralised way for perceptual information to be exchanged and accessed as proposed in Figure 5-3. The bridge stores similar to the cX system, see Chapter 2.2.7, perceptual information about the agent and the state of the environment. That information is not available at the planning level and is currently only intended to remove redundant, or reduce the amount of costly calls to the environment. The bridge, in contrast to a blackboard, only provides access to a domain and problem-specific set of information and no general purpose memory which could be

realised through a hashmap-type data structure. The main strength of the bridge is that it inserts its interface into all instantiated behaviours and offers an uncluttered interface to shared information. Additionally, the approach does not incorporate the idea of perceptual honestly as described by Burke et al. [2001] and implemented in the cX system. Thus, the system allows full access to the environmental information and the designer and programmer can decide which information to use. The focus with POSH-SHARP is on being a flexible, light-weight architecture and hiding information should not be handled in the agent system but designed carefully.

In the next chapter, we will have a closer look at another form of memory which was developed to support interest and motivation modelling for Behaviour-Based AI, see Chapter 2.2.2.

5.3.4 Monitoring Execution

In Chapter 3.6 users of three IVA architectures described the need for logging and debugging functionality which is absent or needs enhancement in their respective systems; this also includes the previous POSH systems.The usage of such functionality would, according to the users, aid the understanding of the execution flow and support the identification of potential problems, both on the design level and the program level. The problem described by the users is that when developing complex agents, the agent is not always crashing or stopping when problems occur. With increasing complexity it becomes harder to tell apart intended behaviour from faulty one[6]. Additionally, the usage of a software debugger, included in most Integrated Development Environments (IDEs), is not always ideal because it pauses the application for inspection which is undesirable for understanding IVAs. To identify mistakes during the execution, POSH-SHARP offers live logging using a logging interface deeply integrated into the POSH-SHARP *core*. The logging uses an internal event listener which receives events from each POSH element that is executed. The events contain a time code and the result of triggering the element. From the developer, this procedure is completely hidden to reduce the amount of visible code they have to touch and memorise. Nonetheless, they can access the log manager and add extra information which gets stored in the log. To allow the easy extension of different developer needs, the log management can be altered using a pre-compile statement for the *core*. This allows currently to switch between two modes of logging. The full log support using LOG4NET[7] or no

[6]This issue leads game developers to be cautious when using new approaches or approaches which allow for learning.

[7]Apache's Log4Net provides a standardised, configurable monitor support in the form of a modular logging architecture. Using XML based configuration files it is possible to set up monitor logs handling even large amounts of data. Log4Net is a dynamic library for the Microsoft *.Net* architecture. It is

logging which is useful for distributing the core with a final product when recording large amounts of data is undesirable.

The log structure uses a millisecond time-code and logs the entire execution in the following form for all agents a_i:

$$S(t) = [t] \quad L \quad a_i.plan(DC(t, a_i)) - return(e(t, a_i))$$
$$plan(DC(t, a_i)) = top(D_{active}, a_i) = e(t, a_i)$$

The drive collection (DC) has only one drive active (D_{active})for each agent a_i at any given time and the Drives maintain an execution stack over multiple cycles. L identifies the log mode which is currently active the modes include: INFO, DEBUG, ERROR.

To limit the stack in size Bryson [2001] introduced the slip-stack. At each cycle, the slip-stack removes the current element ($top(stack, agent)$) from the execution stack and executes it, replacing it with its child, which upon revisiting the drive in the next cycle continues with the child node instead of checking the parent again. This method reduces the traversal of the tree nodes drastically and fixes the upper bound of the stack. POSH-SHARP integrates the same concept but instead of maintaining a stack an internal tree representation is kept and the execution shifts further down the tree when a drive is called.

As the plan unfolds and elements get revisited the log incrementally represents the execution chain of the POSH tree such as the first line will be the check of the goal for the drive collection, the second line contains the check for the highest priority drive and so on. The action and sense primitives are referenced in the log by their canonical method name including the class namespace. This allows for the identification of methods including their annotation name and version number.

The time resolution of the logs can be adjusted based on the developer's needs but to monitor a real-time plan for games, it grows quite quickly due to the fast call times within the tree. A log for the extended StarCraft from Chapter 4.6 reaches a reasonable size[8][9] in minutes. To be able to analyse multiple runs of an long execution, POSH-SHARP writes a continuous-rolling log[10] to manage the individual file sizes better, and it additionally creates a parallel "current" log file which is replaced each time POSH-SHARP get launched again.

The new logging mechanism has a low computational footprint allowing it to log

available at: `https://logging.apache.org/log4net/`

[8]For StarCraft, the game updates the environment with 25Frames per Second (fps). The planner, at each update, performs multiple cycles which result in the same number of log lines.This results in around 50MB after 5 minutes of logging.

[9]An example log can be found in appendix in Figure D-1, page 237.

[10]A rolling file log creates a list of files by adding a numerical identifier to the original file name once a file reaches a certain size.

large amounts of data without impacting the performance. It offers a way to understand the arbitration process by going through the logs line by line. Due to the standardised format, the processing of the logs can be automated or streamed to other applications for a live representation of the agent's reasoning process. The StealthierPOSH game offers a way to visualise the reasoning process by outputting the goals of all agents in the log format on screen[11]. This visualisation and other forms of using the log provide potential directions for future research.

5.4 Concluding Advanced Authoring Support

This chapter presented a novel, project-oriented alteration to the existing Behaviour-Oriented Design by Bryson [2000b]. The focus of the new methodology is to provide better separation of design and programming and to support the development of artificial agents in teams of multi-disciplinary authors. The two case studies and the feedback retrieved from interviewed authors discussed in Chapter 3 create the basis for the newly introduced process steps of the methodology. This new process allows designers and programmers to distribute their work better while still following keeping the project progress in mind. Agile Behaviour Design reduces the dependencies of the different user groups. To further aid the development and to focus on multi-platform development a new arbitration architecture was proposed—posh-sharp—which extends Bryson's original concept of poshand extends it by four new features: multi-platform integration, behaviour inspection, behaviour versioning and the *Behaviour Bridge*. The architecture similar to the original concept of posh still follows the idea of providing a light-weight, flexible and modular approach to designing cognitive agents but increases the usability of the software by reducing potential problem points.

[11] An illustration of the visual logging mechanism in StealthierPOSH is available in Figure D-3, page 239.

POSH-SHARP Contributions:

- POSH required the developer to maintain a large number of source files in nested folders. POSH-SHARP introduced the behaviour library DLL, the core library and the launcher, which reduces the number of files to three and creates an easier to maintain project.
  [simplification]

- POSH-SHARP automatically inspects library files extracting all behaviours and behaviour primitives requested by an agent. This reduces the impact of typos or wrongly associated/non-existing primitives in behaviours.
  [robust primitive loading]

- POSH-SHARP introduces a modular logging and debugging mechanism which allows a developer to trace the flow of information through the POSH graph. This supports the mentioned need of the interviewed authors and can be used to aid the understanding of the action selection mechanism.

- The setup of POSH on developer machines has been simplified tremendously by not requiring any external libraries or APIs to be installed. The setup of JyPOSH is affecting the usability of POSH to a large extend and aligns with the observed trend in the industry to favour approaches which are easy to setup and use.

- The internal mechanisms such as the *Behaviour Bridge* and the *behaviour versioning* increase the capabilities of POSH and remove inter-dependencies between behaviours and support robust incremental to changes to behaviours.

The combination of POSH-SHARP and Agile Behaviour Design is intended to support novice developers by guiding their design but it also allows expert developers to profit from explicit design steps which can be used to verify the progress of a current project.

In the next chapter, an augmentation for behaviour-based action selection mechanisms is presented. The mechanism was implemented in both POSH and POSH-SHARP and allows the design of non-deterministic agents. The presented approach can be generalised to other behaviour arbitration systems and is designed as a "white-box" solution which requires no initial adjustments but allows inspection.

Chapter 6

Augmenting Action Selection Mechanisms

In the previous chapter, a new process model for developing behaviour-based AI for games was presented—Agile Behaviour Design. The approach extends Behaviour-Oriented Design (Bod) by Bryson and Stein [2001], it focuses on strengthening the support for multi-disciplinary teams and industrial development processes, including a more rigorous separation of tasks. In addition to the new process, a new agent framework is presented employing Agile Behaviour Design—posh-sharp. The framework focuses on mobile and platform-independent development and includes the requested tool support from Chapter 3.

In this chapter, a general augmentation of action selection mechanisms is presented—the extended ramp goal model. The augmentation was tested in both Parallel-Rooted Ordered Slip-Stack Hierarchical (POSH) and posh-sharp and evidence for its effectiveness in handling noisy environments and conflicting goals are presented. The importance of handling noisy environments symbolises the impact of unanticipated user interaction and how well agents can handle situations they were not explicitly designed for. This ability to deal with noise introduces non-deterministic behaviour which aligns with deeper agent behaviour, an aim of most Interactive Virtual Agent (IVA) designers.

6.1 Contribution

In this chapter, I demonstrate the application of a biomimetic augmentation to the selection process of behaviour arbitration systems by integrating research on the Basal Ganglia into the process. The research was presented at the *IEEE Conference on*

Computational Intelligence and Games (CIG) by Gaudl and Bryson [2014] and has been submitted to the *APA Newsletter on Computers and Philosophy*. The chapter is purely based on my research on understanding and augmenting decision-making from a cognitive perspective. My contribution to this work is 100%.

6.2 Introduction

This chapter presents a mechanism that addresses the issue of responsive and flexible action selection for behaviour-based AI (BBAI) [Brooks, 1986] or similar approaches to light-weight modular cognitive architectures. The work specifically addresses systems dealing with multiple, possibly conflicting, goals such as the ones described in Chapter 2.2. A special focus is put on a sub-group of those systems that face resource constraints such that they are not able, or not intended, to use a fully fledged cognitive architecture such as SOAR [Laird et al., 1987] or ACT-R [Anderson, 1993]. Limited CPU cycles, restricted memory size, or low power consumption are only a few examples of the mentioned restrictions. In addition to technical resources such as graphical assets or special software, other important and expensive resources are authoring, development and testing time, especially in industrial contexts. To demonstrate and allow for a better understanding of the approach, implementation details, as well as the results of an evaluation carried out in the *MASON* simulation environment[Luke et al., 2005], are used to support the approach's properties.

To clarify the type of problem that is addressed and to give inspiration for its solution, let us start with an example which could take place in a generic role-playing or strategy game. Deciding and maintaining logically sound behaviour or DEEPER AGENT BEHAVIOUR is crucial in games. According to Murray [2004], maintaining the suspension of disbelief is of great importance to players.

Example Scenario (guard in warehouse):

> A player controlling a thief is trying to break into a guarded warehouse. The guard can perform behaviours associated with three major goals, *patrol*, *attack* and $extinguishfire$. The player is moving towards the warehouse and observes the guard patrolling the entrance. The player moves closer to the warehouse. Trying to lure away the guard, the player finds a way to set the back door of the warehouse on fire. As soon as the fire starts, the guard switches to $extinguishingfire$ — this is triggered based upon the game designers' concept for her. The player tries to sneak around the guard but fails as she spots the player while he is moving towards the

back entrance. The guard switches the active behaviour from *patrol* to *attack* because she spotting the thief. The player now runs away, chased by the guard. After some fighting, the guard kills the thief/player, then switches back to *patrol* as no imminent active threat is visible.

Yet, what happened to the fire the player started? The back door of the warehouse is still on fire. After attending for a long period of interactions with the player the trigger signal for $extinguishfire$ was removed from the stack of sensory information for the guard. A naive solution would be to let the trigger remain on the stack indefinitely and for this simple example, it seems a feasible option. However, scaling up the problem to a large set of agents and triggers and not removing stacked triggers is impossible and even distinguishing which triggers are still of importance is a difficult problem, requiring additional computational resources.

The main point is, in large design spaces it is hard for a designer to keep track of all possible scenarios and inter-dependencies of behaviours. Additionally, designing game agents that behave in a believable and concurrent way is already a complex task. Due to the large size of current games and their underlying control structures, it is non-trivial to keep track of the maintenance and inhibition of timed actions. In digital games, it is quite common to allow the AI only to occupy a fixed small number of cycles per frame as most of the computational resources are needed for the graphic representation. Including a heavy-weight cognitive system to control multiple agents into such an environment is in most cases not desirable as the cognitive architecture requires both more CPU time, and also more time to design. Additionally, designing the specific cognitive agents themselves is generally more time-consuming than the typical static approach to game characters. For IVA which only need to give the impression of DEEPER AGENT BEHAVIOUR, cognitive abilities are usually not necessary. In the above example, a designer—similar to a writer—would create a story about what the guard should do and how she should react to certain stimuli. Removing this creative process would either result in a huge impact on the players' immersion or it would require an enormous amount of computation to do meaningful story planning. Mateas discusses this further in his work on *Façade* and ABL, see Chapter 2.2.8. Game-play designers specialise in creating human-understandable situations, reactions and characters. Despite promising research by Mateas [2002], automating this whole creative process is currently far beyond the current state-of-the-art in dynamic planning and story generation. The present main interest of game AI designers and engineers it to have flexible, modular tools for creating template agents and then modify those to create the desired outcome [Grow et al., 2014].

The work presented here is motivated by an analysis of existing agent architectures and agent modelling environments for autonomous agents in digital games [Grow et al., 2014], see Chapter 3. Existing cognitive approaches such as SOAR, ACT-R, LIDA [D'Mello et al., 2006] and CRAM [Beetz et al., 2012] are extremely powerful, allowing the creation of sophisticated agents. However, due to the high complexity and steep learning curve, they are seldom used outside of academic demonstrations and simplified problem spaces. Even where they are used, they are primarily employed in communities strongly linked to an academic environment, such as military war games. When full cognitive reasoners or large expert systems are not needed or applicable, lightweight architectures and models such as Behaviour-Oriented Design by Bryson and Stein [2001] or purpose specific architectures such as Pogamut [Gemrot et al., 2009], Mateas and Stern [2002] can be used. Purpose-specific architectures offer an optimised work-flow for specific settings, reducing development time. Chapter 2 offers a more detailed discussion of the structure and existing applications of those architectures.

Lightweight systems, due to their lower additional computational cost and lower learning curve are generally more favoured in the non-academic application[1]. To date, these systems have been used most widely in the computer games industry, a substantial part of the contemporary economy that takes in more money than more traditional entertainment such as the film industry. Game AI requires agents that are able to act in real-time, can be instantiated quickly and leave the impression of human-like or animal-like intelligence. Lightweight cognitive architectures may be used either for individual agents or for swarms of shallow agents in a variety of digital environments (not only games), as well as for small autonomous robots such as Aibo or Roomba, or even substantial numbers of swarming robots [Chaimowicz and Kumar, 2007; Rubenstein et al., 2014]. Due to the flexible nature of the applied approaches, the resulting system can be tailored towards a specific scenario, reducing the computational cost drastically. This contrasts with most cognitive architectures which are intended as general problem solvers applicable to a wide range of problems. To better facilitate developers and researchers using lightweight architectures, and to enrich their action selection and behaviour arbitration mechanisms, biomimetic models have been examined.

The mechanism which is presented in this chapter is based on a ramp function, similar to the ramp activation found in the mammalian brain cells responsible for goal switching. The model is based on a system of dopaminergic cells in the Basal Ganglia of the mammalian brain [Cools, 2012; Brown and Nee, 2012]. The model is designed to apply to a broad range of systems. In keeping with lightweight architectures, it has a low

[1] In digital games, for example, the designer is typically not interested in having hundreds of cognitive agents but just the impression of plausible actions for groups of those IVAs and intuitive means to design them.

computational overhead, making it highly versatile. The final model allows for an easy way to control the maintenance, inhibition and switching of high-level behaviours in cases where static linear goal structures or predefined behaviour switching is undesirable for the action selection mechanism.

Action selection is a key element of cognition, and even lightweight models such as those discussed here can provide new insights. First, evolution also favours lower costs, so sometimes insights into nature can be gathered from the experience of engineering, despite the differences in implementation between massively parallel biology and essentially sequential silicon, Bryson [2005].

Even models of consciousness do not necessarily need to be implemented at the neural level [Franklin et al., 2009; Bryson, 2012]. Second, lightweight architectures operate at a level of abstraction more similar to the philosophy of mind, so are more easily comparable. Finally, more agile and accessible AI development allows more exploration of theories and their consequences.

Here, an emphasis is put on the creation of lightweight yet cognitively-motivated models to observe and perceive the world in sufficient detail to react or behave naturally. The approach uses a functional representation to model phenomena which are expressed in a similar way in the neuronal structures of the mammalian brain. It focuses on the outcome of modules—both the actual brain structures and the functional modules—and aims to represent the functional outcome accurately, rather than the actual underlying structure. As computation resources get more available, in terms of numbers of CPU cycles the AI is allowed to use, a shift in the usage of approaches towards more closely representing the underlying structure of the cognitive agent is possible, although nature, of course, has had billions of years to create powerful structures such as the brain, and its integral host the human body. In digital agents and robotics, most of the organic structure evolved by nature is not needed to create similar outcomes. Thus, it is possible to focus on creating modules which are functionally similar instead of mechanistically similar.

In the next section, the current research on biomimetic models and their applicability to behaviour arbitration is described, introducing the extended ramp goal model—ERGo. Details and a code example on how to integrate the approach into existing arbitration mechanisms are given as guidelines for further development of the integration of the model into other systems. To support the presented argument, the results of an evaluation performed in a real-time, game-like simulation environment using a previously-published system as a baseline for comparison are given. The chapter concludes with a discussion on the impact of different parameters on the model, next possible steps and possible extensions to this work.

6.3 The Extended Ramp Goal Model (ERGo)

In this chapter, we will have a closer look at the biomimetic mechanism and its implications on scalable behaviour arbitration. As a starting point, the motivation for applying biomimetic concepts to action selection schemes are presented. This motivation in combination with an analysis of cognitive goal selection and maintenance leads to the main drive for the ramp function arbitration mechanisms.

6.3.1 Approach: Biomimetic Models

Rohlfshagen and Bryson [2010] introduce Flexible Latching as a mechanism to handle multiple competing goals. Flexible Latching starts from a simple latch, see Figure 6-1, which reduces *dithering*—a rapid switching between goals. When dithering, more time is spent transitioning between goals than in their useful pursuit and consummation. Without a latch, a goal executes once the trigger condition is met and stops immediately after that. A latch thereby acts following the same principles as a hysteresis function.

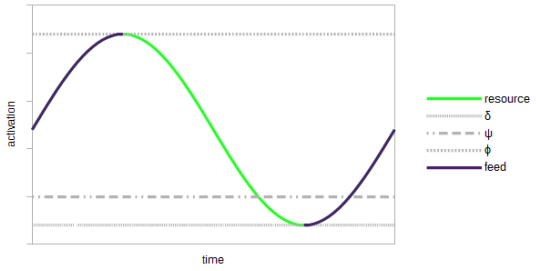


Figure 6-1: A Flexible Latch using two thresholds—δ and ϕ—to control dithering. In a simple latch, a goal can take control from when activation reaches the lower boundary δ until is reaches the upper boundary ϕ. Reaching ϕ, the goal is inhibited until activation falls below δ again. A flexible latch adds a third threshold, ψ, above which a latch is recomputed if the agent is interrupted. Rohlfshagen and Bryson [2010] find the best threshold for ψ to be $\psi = \delta$.

For the sake of an illustration, imagine the following example:

> A leaking canister loses water over time. As soon as a low water level—threshold ϕ—is reached, the canister is filled up again to that level. If it is only refilled up to ϕ whenever the water is below ϕ, the time between each re-fill is relatively short. A strict latch now adds another offset δ on top of the lower threshold. Now, whenever the water reaches ϕ the remaining actions are spent to refill the water until is reaches the higher level, threshold δ.

Such a latch is useful under the assumption that it takes time to start and complete an action. The strict latch allows this extra time between δ and ϕ which can be spent on alternative actions. Flexible Latching extends the Strict Latching by dealing with interrupts and re-evaluating whether the current goal should still be pursued. Rohlfshagen and Bryson [2010] demonstrate this approach to be more efficient than a simple latch as the agents do not pursue goals that are neither urgent nor convenient after the interruption.

Now taking a closer look at other selection processes inspired by nature, neural networks (NN) are the most prominent, flexible selection mechanisms. By using an artificial neural network (ANN), it is possible to learn and solve selection tasks for problems where an algorithmic description of the problem is not known or is costly. ANNs are able to approach general solutions only by providing them with a set of specific, known input and solution pairs to adapt them towards the solution space. However, for ANNs the overall action selection or computational process is not transparent, thus tweaking a Neural Network to perform in a certain way is difficult, see Chapter 2. Also, NNs are typically trained to solve static problems rather than a continuous problem like action selection or real-time behavioural control with changing or unseen situations. An example of a commercial game using a neural network is Black&White by Lionhead, which uses a neural net for training a few aspects of the player's pets intelligence.

Taking a look at a single neurone model reveals some interesting mechanisms which can be exploited in other contexts as well. There exists a variety of activation functions for neuronal models. Those include the spike or DIRAC function used in Spiking Neural Networks (SNN), the sigmoid which has a fixed output range between zero and one, and the ramp function which combines a monotonic increased activation and an instant activation drop. This last forms the basis of the model presented here.

Biomimetic models like ANNs are an important asset of the computer science toolset. They offer useful and scalable solutions for addressing complex issues. Redish [2012] indicates that the ramp function is favoured for goal arbitration, which will be detailed in the following section. This finding motivates the present approach as it is a straightforward and elegant mechanism for augmentation.

6.3.2 Basic activation mechanism

Two defining features of the exhibited ramp-like activation in the brain are a linear growing activation and a rapid activation drop, see Figure 6-2. The hypothesis that brains exploit ramp functions to arbitrate between high-level goals is utilised as the basis for the light-weight arbitration mechanism, ERGo. In contrast to most ramp

function related selection approaches [Stewart et al., 2012; Velsquez, 1998] that apply ramp functions in the context of neural networks, the current approach is the first attempt to use a ramp-like criterion directly in a behaviour-based arbitration process without using a neural network to control the maintenance. The presented models use a strictly monotonic activation gain and an instant activation drop when reaching the success criteria for the goal, this provides a predictable and visual, yet, flexible mechanism.

A generic behaviour-based action selection mechanism is used to illustrate how the extended ramp works. For a given set of behaviours[2] $B = \{B_1, \ldots, B_m\}, m \in \mathbb{N}$ a set of goals $G = \{G_1, \ldots, G_m\}$ and ramps $R = \{R_1, \ldots, R_n\}, n \in \mathbb{N}, n \leq m$ is introduced. R_a is the ramp for B_a and G_a is the goal which B_a is trying to satisfy, $a \in \{1, \ldots, n\}$. The additional behaviours $B_b, b \in \mathbb{N}, b \leq m - n$ try to satisfy goals G_b without being augmented with a ramp. Each time step t R_a adjusts its activation based on the Boolean activation state $\alpha_a(t)$ of the behaviour B_a, the Boolean urgency signal $\upsilon_a(t)$ and the stickiness $\omega_a(t)$ of the behaviour. All ramps share the same increment i and activity multiplier μ which define the accumulated activation in the following way.

$$R_a(t) = \begin{cases} R_a(t-1) * \mu & \text{if } \upsilon_a(t) = 1 \\ R_a(t-1) + i & \text{if } \alpha_a(t) = 0 \\ R_a(t=0) & \text{if } \alpha_a(t) = 1 \wedge \omega_a(t) = 0 \\ R_a(t-1) + (i * \mu) & \text{if } \alpha_a(t) = 1 \wedge \omega_a(t) > 0 \end{cases}$$

The influence of an active behaviour on the activation is presented by $\alpha_a(t) = 1$ and $\omega_a(t) > 0$. This results in an activation modified by the activity multiplier μ.

$R_a(t) = R_a(t-1) + (i * \mu)$

The increased activation is supported by the work of Redish [2012]. He states that the goal cells in the Basal Ganglia have a higher firing rate when a related goal is pursued. Even when a behaviour is not active, it still gains activation.

$R_a(t) = R_a(t-1) + i$

The combination of these two mechanisms removes most of the requirements of needing a direct binary switch for the behaviours to arbitrate successfully. This combination minimises the direct competition between behaviours as well and increases the robustness of the action selection in cases of noisy switching signals.

Thus, the approach contrasts the currently available selection principles in games. These existing mechanisms heavily use binary triggers because they are initially easy

[2]Note that we use *behaviour* here to refer a collection of actions, senses and other cognitive states necessary for achieving a particular goal. In many architectures, behaviour decomposition is actually orthogonal to goals—one action can serve multiple goals.

to implement and understand. Another method which is used and initially looks similar to the ramp are utility functions, see Chapter 2.1.1. The main difference between the approaches is that utility functions in games normally optimise a specific criterion expressed by an Interactive Virtual Agent whereas the ramp models a cognitive process internal process. The strength of utility models is that it is possible to pick a different function and alter it to fit the described behaviour. The inclusion of various mathematical functions and the analysis of how they perform in a particular situation requires a large amount of parameter tuning. The ramp is intended to work in opposition to a complex approaches that require problem specific tuning by providing a minimal, "as-is", white box approach with initially no tuning.

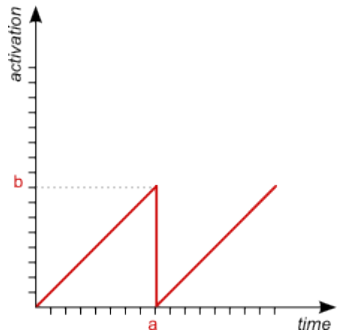


Figure 6-2: A single ramp function used for inhibiting a behaviour. A behaviour controlled by a ramp is only inhibited when another behaviour gains a higher activation or once its goal is reached. Once the goal is reached, activation instantly drops. The behaviour completes its goal at time a, with a certain activation b.

To allow the agent to influence whether a behaviour needs to be urgently triggered, the agent is able to trigger the urgency signal υ. Upon receiving the signal, the ramp amplifies its activation using the activity multiplier μ—a percentage based influence on the global action selection. Using μ for urgent execution results in an exponentially growing activation level. An example for an amplified behaviour is *Behaviour3* in Figure 6-3 which is triggering υ_3 at $t = 41$. For the experiments, μ was set to a value within the range of 1.0 and 2.0. If $\mu = 1.0$, activation is not affected by the urgency signal at all. If $\mu = 2.0$, the activation is increasing quadratic. The impact of negative urgency on agents has not yet been investigated. Negative urgency would be reflected by $0 < \mu < 1.0$ and would result in a decay or dampening of the activation level. If B_a needs to urgently execute, υ_a is set to *true*. This indicates the need for a rapid behavioural change. The result of using the urgency signal υ is inspired by natural phenomenon inside the mammalian brain, where it takes a small amount of time for the activation to spread before even urgent actions are executed. However, the time span between the trigger and the execution of the behaviour is short. In the case of the

experimental setup, the time span is defined by the unit count of one of the resources and its amount sinking below δ_L.

One of the aims of the general behaviour arbitration augmentation—ERGo—is to simplify the action selection process. To achieve this simplification, the work focuses on a low coupling of the ramp goal model and the rest of the agent as much as possible. Additionally the number of parameters which have to be adjusted for a working integration should be kept as low as possible. Thus, the parameters are limited to υ, an urgency signal, and μ, which amplifies the activation of our model. Using only these asynchronous signals, there is no need to include problem-specific components like agent specific resource properties in the control. This makes ERGo easier to comprehend and integrate with other architectures as the properties should generally be handled directly by the behaviour primitives.

6.3.3 Duration of activation

Action selection requires both recognising when to start a goal, and also how long to pursue it. In ERGo, a goal and its associated behaviours become active when one goal's activation is higher than others. Activation continues building until another threshold is reached and then it drops to zero (see Figure 6-3). This duration is controlled via the stickiness ω of goals as part of the mechanism. This was also inspired by mammalian behaviour when animals feed after a period of reduced available resources; they do not stop feeding even if their stomach has reached its capacity. This is referred to as binging [Mathes et al., 2010, 2009]. However, just as with the latch, performing behaviour for enough time to build up reserves should be viewed as a normal part of action selection. For an active behaviour B_a, once its goal conditions are met—$\alpha_a = 1$ and the agent has accumulated enough resources of one type to reach δ—the stickiness is decreased until is reaches zero. During this time, the behaviour still accumulates activation. In other words, the agent—even though the goal G_a is met—continues to pursue G_a until $\omega_a = 0$.

$$R_a(t) = R_a(t-1) + (i * \mu)$$

The only way to interrupt this is either by having a higher activation due to an urgency signal or due to an environmental interrupt which disturbs the current behaviour and resets the activation to the lower boundary. Both phenomena are also present in nature. For example, an animal is feeding and a predator jumps out of cover. If the current feeding behaviour is not instantly interrupted, the animal would simply die. The stickiness ω of ERGo is similar to a latch but is encapsulated within ERGo. Its purpose is to allow the agent to handle environments where resources are sparse. It is

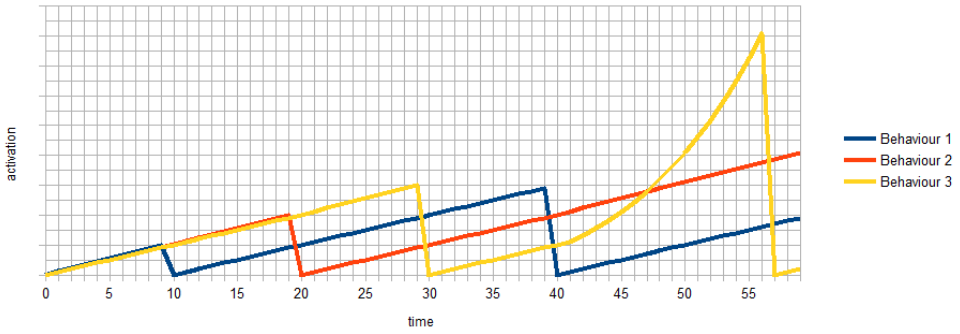


Figure 6-3: Internal activation levels of three behaviours using ERGo. From time $t = 0$ to $t = 9$ and $t = 30$ to $t = 39$ *Behaviour*1 is active having a higher activation. At time $t = 9$ the success criteria for the first behaviour is met and the activation drops resulting in the activation of the second behaviour. *Behaviour*2 is active from $t = 10$ to $t = 19$ where its goal is reached. As all behaviours have the same inclination, they automatically schedule into an activation pattern. At $t = 41$ the urgency signals is triggered for *Behaviour*3 resulting in an exponential gain of activation and an activation at $t = 47$.

part of the internal model and hidden from the agent to allow for an easier integration, minimising the parameters exposed by the agent with the aim to reduce the cognitive load of a developer during design time.

For the experiments, the resources of each agent were initialised using a random value within the bounds of δ and ϕ. This minimises not only possible direct conflicts at start-up time, but it also provides more activity and liveliness to the simulation. The latch is modelled using the internal motivational states of an agent, for example, hunger or thirst. In contrast, the ramp uses a hidden internal counter of its activation, a signal for urgent execution of the integrated behaviour, and a signal for when the behaviour achieved its goal. Those mentioned signals dealt with asynchronously. Thus, ERGo's integration does not introduce additional conflicts by waiting for a trigger signal. This allows us to run the computation of the ramp and its augmentation on even large sets of behaviours in parallel.

6.3.4 Integration

Following the description of ERGo, the integration of ERGo into a particular agent model and simulation[3] is presented next. The description of the extended ramp so far has primarily focused on explaining the mechanism of a single ramp. The interaction between multiple ramps is managed within the execution frame of each augmented

[3]The simulation itself is discussed in the subsequent section.

behaviour. Whenever a behaviour tries to gain the control, the system validates if other behaviours have a higher activation—a mechanism similar to the selection process in the Basal Ganglia. If those behaviours with a higher activation can execute, they will suppress the behaviour trying to gain control. Thus, there is always only one behaviour $B_n \in B$ of augmented behaviours active. Due to this restriction, the selection is always conforming with the rest of the underlying hierarchically ordered action selection mechanism without overriding the general priority scheme.

To aid the understanding of the selection process the action selection mechanism by [Bryson and Stein, 2001] is used, discussed in Chapter 2.2.9. Due to the modular nature of POSH, ERGo can be integrated as an additional sub-component into the action selection mechanism without having to change large portions of existing code or the general action selection scheme. A similar integration into a prominent game approach such as BehaviorTree, discussed in Chapter 2.1.1, is possible based on the similarity of POSH and BehaviorTree (Bt). To allow for a better comparison of the results, the original Flexible Latching code-base by Rohlfshagen and Bryson [2010] has been modified. By extracting the Latching code from the behaviours, it has become possible to integrate the resource storage, *_energy*, and its adjustment back into each of the behaviours. This extraction makes the whole code more transparent without changing the functionality and exhibited behaviour. A benefit of this refactoring is, the how and when of resource accumulatation is more visible, see the behavioural action *a_drink* in Code 6-4. The refactoring was required because beforehand, the behaviour had no internal representation of its resource state other than the latch which internally contained the problem specific parameter, thus, "hiding" important information from the user.

The agent's action is split into three distinct parts, see figure 6-4. The first part–line 1 to 8–is responsible for environmental interrupts. The simulation environment controls when and how to trigger those interrupts. If the ramp should reset the activation, it is calling $GoalCell.reset(self)$ for each augmented behaviour. This `reset()` results in a re-evaluation of the internal activation. The second part until line 16 is responsible for either leaving a food patch when it is empty or telling the agent to feed on the resource patch.

The last part in Figure 6-4 is referring to the goal criterion which informs an agent that it is finished accumulating resources and that the ramp could release the activation now. The release is triggered inside the *reached_goal* method which is reducing the stickiness and resetting of the ramp once the stickiness is zero.

```
def a_drink(self):
  if self.inter.should_interrupt(self._energy):
        GoalCell.reset(self)
        self.prev_target_loc=self.drink_target.loc
        self.target=None
        self.signal_interrupt()
        self.inter.increase_count()
        return 0

  if not self.target.agent.Resources.s_has_food_left():
        self.signal_interrupt()
        self.target=None
        return 0

 self.target.agent.Resources.a_reduce_food_load()
 self._energy += common_increment

 if self._energy > common_upper:
       self.reached_goal()
       return 0
 return 1
```

Figure 6-4: Python code illustrating the inclusion of ERGo into an existing goal module.

6.3.5 Summary of Augmenting Behaviour Arbitration

Current research on the Basal Ganglia suggests that a ramp-like activation function controls the goal maintenance in the mammalian brain. In this chapter a new mechanism—ERGo—which extends the application of the ramp beyond neural networks to more abstract and light-weight action selection systems. The augmented behaviour is able to react to sudden changes in the environment. The communication between the extended ramp and the behaviour is through a well-defined, sparse signal flow. The implementation is using a low-cost computational model of the ramp and is based on a Python agent using the POSH action selection [Bryson and Stein, 2001].

```
def reached_goal(self):
  if not self._active:
    return
  if self._sticky > 0:
      self._sticky -= 1
  else:
    self._activation = self._lower_bound
```

Figure 6-5: ERGo's *reached_goal* definition, reducing the stickiness if the goal criteria is met.

In the next section, the test domain is described where multiple conflicting goals can arise for an agent. Natural agents, from single cell paramecia to human beings, face this situation regularly, and so should Interactive Virtual Agent. For example, a small child indecisive if he should sleep because he is tired or to continue to play because it is fun. Additional information on the agents' behaviours is presented to allow for a better understanding of the domain and the possible actions of an agent.

6.4 Evaluation

Behaviour-Oriented Design was chosen as the light-weight architecture test platform. BOD allows the description of cognitive agents utilising the parallel-rooted slip-stack hierarchical (POSH) dynamic plan structures. POSH includes a linear goal structure where each goal has a fixed priority with respect to the others, although each goal can be inhibited either by having unmet preconditions or through a system of scheduling. One reason POSH is well-suited for the selected experiments is that it has already been fitted with a modification to this structure to allow more biologically-plausible action selection. This mechanism is Flexible Latching by Rohlfshagen and Bryson [2010], described earlier in Section 6.3.2. As a simulation environment, the MASON simulation platform [Luke et al., 2005] is utilised because of the well-defined and easy to configure Java interface and because it offers and easier comparison to previous work using MASON.

The simulation environment is modelled after *Sim1* used by Rohlfshagen and Bryson [2010] to reproduce their case study on the Flexible Latch. The world contains two resource types, water and food, equidistant from the centre of the map in 150 units. The world is 600 by 600 units and the agents start at the heart of the map, see Figure 6-6.

Agents can travel two world units in any direction for every tick of the system clock[4]. The map is wrapped around the horizontal and vertical edges. If an agent travels only in one direction, it will create a circular path around the world. Due to the layout of the map, there is no benefit from moving over the map edges as the distances are exactly the same. It is also noteworthy to mention that an entity in the simulation is unable to block a path, resource, or another agent in any way, which would be possible in nature but introduce unnecessarily complicated dynamics for the task at hand. The only time agents interact is during grooming.

Each agent constantly uses 0.1 resource units of water and food each tick to survive, simulating natural metabolic costs and presenting the problem of self-sustenance. The

[4]To simplify the model, discrete time steps are used instead of real-time calculations which not only allows more fine-grained control it also makes it possible to speed up the simulations beyond real-time.

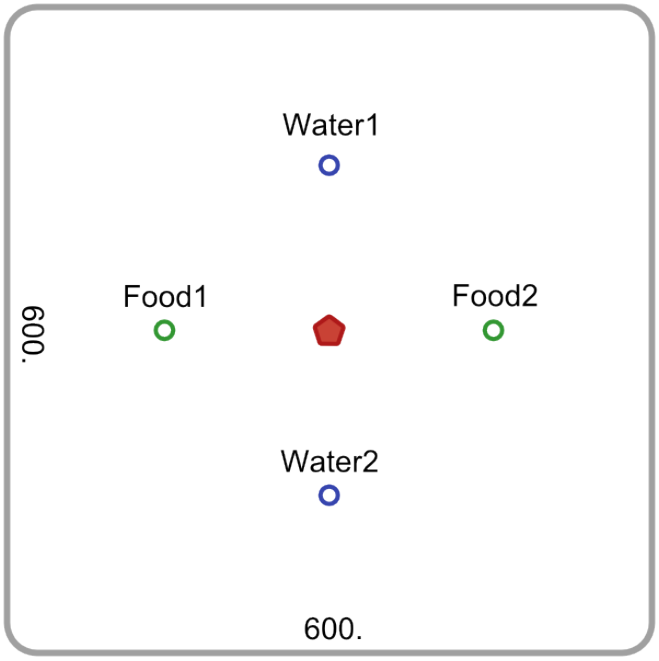


Figure 6-6: Simulation Environment in a Mason agent simulation. The world is 600x600 units. It contains two food and two water sources equidistant from the centre. All agents spawn at the centre at time $t = 0$.

amount of energy needed does not change during the simulation even if an agent does not move. If an agent's accumulated store of one of the two resources drops to zero, then the agent dies. All agents are initialised within a lower limit δ and upper boundary ϕ for the two resources. Whenever an agent is feeding on one the resources, it gains energy, 1.1 units of the resource. The gain is set to be larger than the consumption so that the agent would have a chance of surviving and not continuously lose energy even while feeding. For the experiment, the gain is set to ten times the metabolic cost.

To allow the agent to track when it urgently needs to feed on a resource, its "intelligence" is made sensitive to when its units of a particular resource drop below δ—an artificial threshold used for modelling hunger. Whenever the units reach the upper bound ϕ, the agent is designed to detect that it has satisfied the need for that resource. Once a behaviour is satisfied, it may distribute its time across any other of its goals. The shortest path between one food and water resource requires an agent to spend approximately ten units of both resources which are the amount it can gain from feeding for one tick. For the current experimental setting, the resources are never depleted. Therefore, the agent only needs to take care of consuming enough food and water to stay alive and pursue its other goals. A more realistic setting might include patchy, degenerating resources, but this level of complexity will not be introduced to

the current preliminary experiments because it would only distract from the current objective of analysing noisy signals. At any rate, foraging theory is fairly well understood, see Stephens and Krebs [1986]. This chapter focuses the research on lightweight mechanisms for goal arbitration.

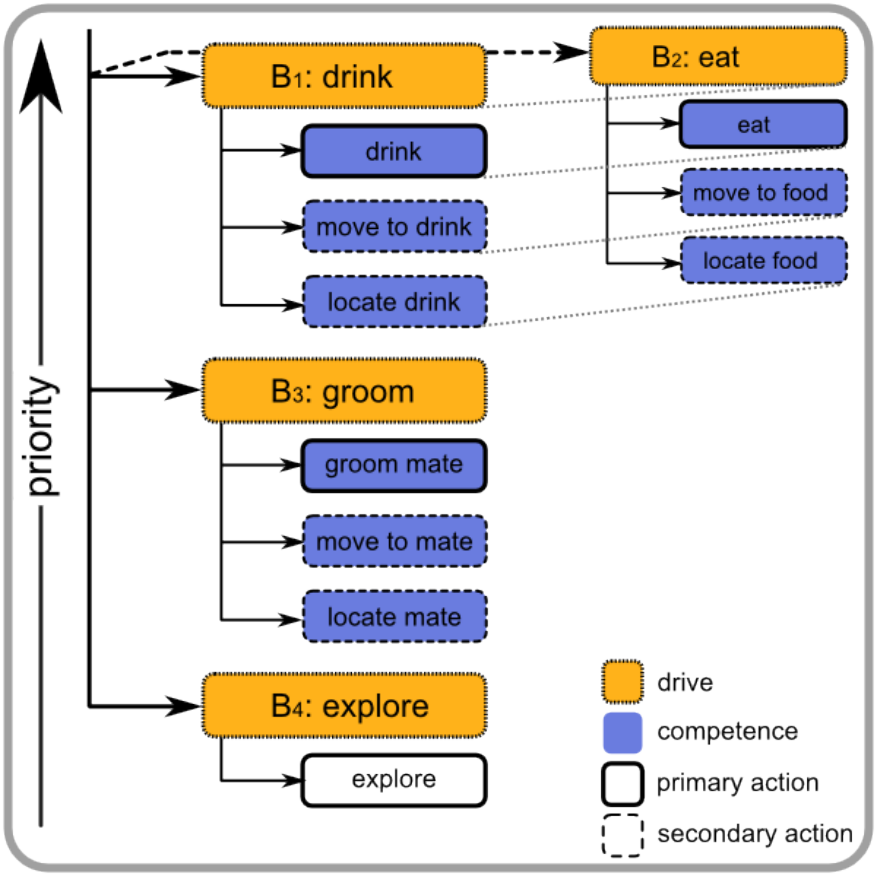


Figure 6-7: A condensed view of a drive collection. It specifies the behaviour of one of the agents in the simulation and contains four behaviour drives, prioritised from top to bottom. Drives B_1 and B_2 have equal priority, meaning they are equally important and their priority must be arbitrated in some sensible manner so both can be achieved in response to a direct internal conflict regarding the resources.

In Figure 6-7 a simplified version of a POSH action plan is given. This plan is used for all agents in the simulation. Each agent has four drives which are prioritised based on each drive's position in the action plan. The higher the drive in the plan the higher its priority. Each drive is designed to satisfy a specific goal of the agent, for example, Drive B_1 represents the need to drink. In POSH, those goals are specified by internal or external senses, in this case, the sense *wantstodrink*. There is a special case which is behaviour B_4– the lowest-priority drive. The lowest drive should always

be able to execute as it is treated as a fallback as well. If no drive can be executed the plan terminates and the agent will stop and terminate as well. The behaviours B_1 and B_2 have equal priority indicating they are equally important to the agent—both are required for its survival. At that point, the biomimetic augmentations are introduced to ensure that both drives are met in an efficient way, with neither dithering nor neglect.

6.5 Results

In the chapter, we looked at the biomimetic mechanism based on research on the Basal Ganglia—ERGo—for augmenting existing action selection mechanisms. The experimental settings and the test environment in which the effectiveness of the ERGo model compared to Flexible Latching were evaluated was presented and discussed to allow for a better understanding of the results.

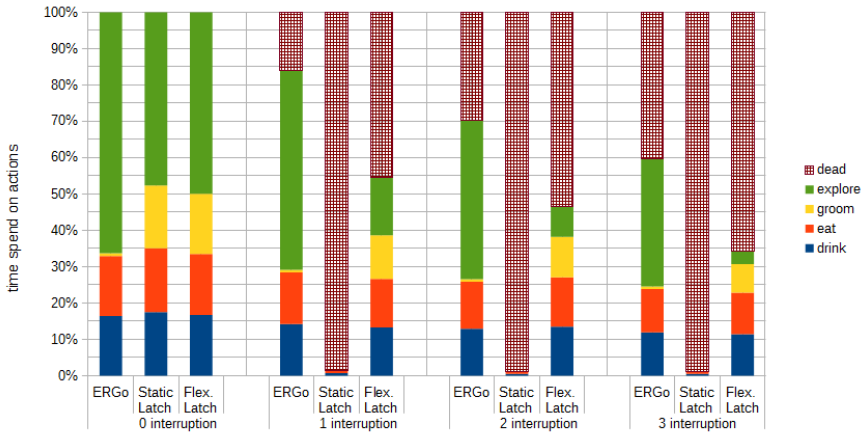


Figure 6-8: Comparing the three behaviour augmentations Static Latch, Flexible Latch and ERGo. Illustrated is the change in invested time for an interrupt progression $i = [0, 1, 2, 3]$. As the interrupts increase the Static Latch becomes unable to arbitrate behaviours appropriately. This results in a high death of agents. ERGo and Flexible Latch are able to adapt to the interruptions. ERGo agents remain significantly more alive.

Fifteen independent trials per parameter were run to analyse the influence of each tested parameter on the augmentation. Additionally, all simulation with the Flexible Latching model were re-run as well to have a direct comparison on the same system. Each trial run was allowed $5,000$ ticks, as in most cases the simulation either converged to a stable state (death of all agents or a stable number of surviving agents) before that time. With an increased number of trials (50 runs) in contrast to the original 15 trials, the system reaches stable results with a low standard error. At first, both

approaches—Flexible Latching and ERGo— were analysed in relation to how well they performed and how well they can handle non-hostile environments. In non-hostile environments, both models perform well. Due to the random initialisation of the resources for each agent's internal storage, the standard deviation for all agents can be quite large. To compensate this deviation, the number of trials was increased to the previously mentioned 50 trials in comparison to the original 15 trials.

The following evaluation criteria were used to judge the quality of a well-performing augmentation:

1. time the agent remains alive,
2. time left for individual behaviours beyond those needed for survival,
3. robustness in face of noise and interruptions, and
4. programmability.

For the experiments, the lower threshold was set to $\delta = 40$ and the saturation threshold to $\phi = 44.5$. First, the augmented agents without interrupts were tested. In all trials for this setting, all augmented agents remain alive, demonstrated in Figure 6-8 by the first three bars. Both Latches invest a fixed amount of time on the two highest priorities, drinking and eating, and then spend the remaining time on lower priorities, grooming and exploring. As exploration does not have any additional requirements compared to grooming, the largest fraction of time is invested in exploring. Grooming and exploring are not life essential to the agent and grooming has the additional requirement of having a grooming partner, thus, ERGo spends far more time in exploration than both latches and less time in grooming. For future work, it might be interesting to introduce a need or motivation for the agent to groom. Additionally, this additional need for grooming would move the simulation closer to actual experiments on social animal behaviour. The presented result is based on the mechanisms underlying the Latch where a *fixed* threshold guarantees that extra time is invested in other actions. However, ERGo's stickiness ω applies a more dynamic criterion resulting overall in more actions to be invested in *all* goals. Additionally, those actions can be interrupted more easily which is visible in Figure 6-9 once the interrupts increase.

As the interrupts increase from $i = 0$ to $i = 3$, the Static Latch is persisting on executing actions which are not advantageous. Flexible Latch is able to handle the interrupts better than Static Latch, visible in the lower death rate. It scales down all actions equally which puts high pressure on the agent, as the life essential actions are also reduced. ERGo scales best, demonstrated by urgent behaviours inhibiting others

from executing when they need to perform instead. Life essential behaviours maintain the highest priority but lower level goals are still pursued.

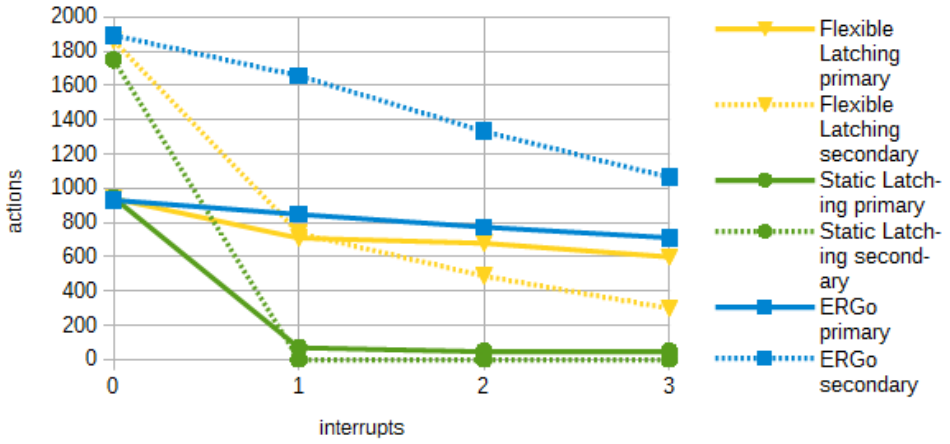


Figure 6-9: Comparing the effects of interrupts on the priority hierarchy of behaviours—demonstrated by comparing total primary and secondary behaviours. The amount of higher and lower priority behaviours is nearly equal for both Latches allowing an equally high proportion of lower priority behaviours to be executed. Once interrupts increase, the Static Latch is unable to remain in a stable state—most agents die. Flexible Latch and ERGo scale down the number of actions when interrupts increase. However, the actions for Flexible Latch are decreasing disproportionate compared to ERGo.

Figure 6-8 illustrates how the differentiation between lower and higher priority behaviours is handled in both, Latches and ERGo. With increased interrupts ERGo and Flexible Latch scale down but ERGo maintains a similar ratio of higher and lower prioritised behaviours.

As ERGo responds only to signals by the agent, it does not optimise free time as efficiently as the hand-tuned Flexible Latch. However, the presented approach minimises the interdependence of particular parts of an agent, thus, increasing the robustness and programmability of it. For the new model, no problem-dependent parameters were specified in ERGo with the goal to allow for a better integration into other action selection mechanisms and the creation of a general purpose augmentation.

The focus of the current experiments was mainly on noise in the decision process and especially on interrupts. This focus allowed the analysis of how well an agent is able to handle non-scripted situations, e.g. unpredicted player interactions in a game. Increasing the interrupts is in some ways similar to players probing or testing an agent or system by trying to find a way of breaking it. In heavily scripted games or full information games, the agents are typically not affected by such attacks. However the

more agency, dynamic planning and uncertainty is introduced into games, the easier it is to break the agents due to the need to react to different stimuli depending on the situation.

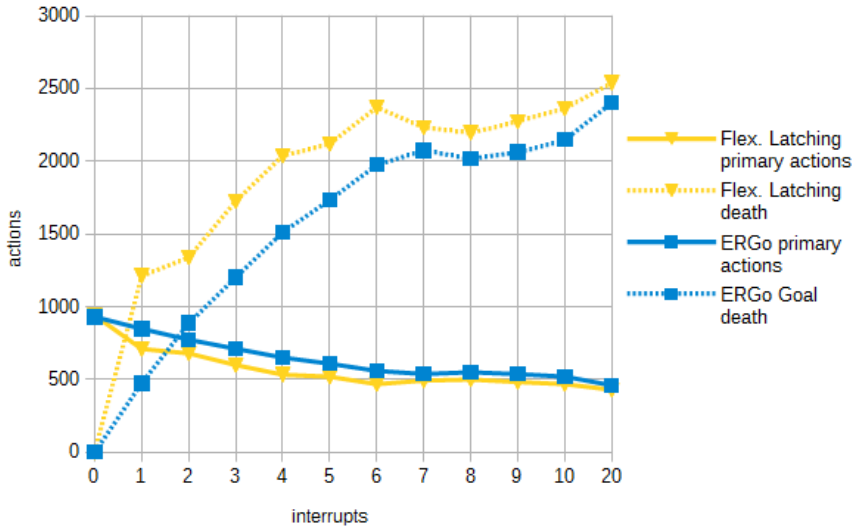


Figure 6-10: Influence of increasing numbers of interruptions on death rate and executed primary actions—eating and drinking—for Flexible Latch and ERGo. Flexible Latch presents a higher death rate in all settings. For the number of high priority actions, ERGo and Flexible Latch start equally, around 1000 actions. As interrupts increase, ERGo performs more high priority actions until 8 interruptions per successful behaviour .

The results of a further interrupt increase are presented in Figure 6-10. The graphs illustrate the influence of interrupts by using a linear increment from 0 to 10 and a final increase to 20 interrupts to understand if some significant changes or converging behaviour is emerging.

Two interesting observations are possible from the figure. The first is the point where death rate and primary actions cross for each augmentation. This point indicates a shift in the agent behaviour where on average the agent loses much activity and liveliness. For Flexible Latching, this point is before one interrupt per goal attempt. ERGo reaches the same situation at two interrupts. This difference suggests that ERGo augmented behaviours at least in the observed experiments are more resilient in terms of interrupts. The second observation supporting the previous suggestion is that, while ERGo is performing a similar amount of primary actions per simulation, the death rate is always a considerable amount lower than for Flexible Latching. Additionally, there is also a larger number of secondary actions ERGo performs. It can be argued that a change of latch size or the lower threshold δ could compensate for that.

However, the central point to emphasise here is that hand-tuning can also be done for ERGo when modifying the stickiness of goals or the activity modifiers.

As soon as disruptions and interrupts were introduced into the agents' behaviour maintenance process, the static latch agent has a considerable action drop even by introducing only one interruption per behaviour. The Flexible Latch instantly cuts to a third of its actions. With increased noise and hostility of the environment the biggest advantage of ERGo becomes visible. The whole range of executable behaviours scales in a way that allows more agents to stay alive, active and lively—demonstrating the full range of possible behaviours. The ERGo action activity is only dropping by a tenth compared to the noise free setting. The more interruptions are introduced, the bigger the impact on the death rate of the latched agents whereas a significantly higher percentage of the ERGo augmented agents remain alive.

Summarising the results: In this section, experimental results from evaluating the extended ramp goal model—ERGo—are presented. Those results were compared to a similar biomimetic approach—Flexible Latching [Rohlfshagen and Bryson, 2010]. The experiments stressed the ability of both methods to handle noisy action selection based on interrupts in the selection process. A particular focus was put on environments where action selection was already difficult. At the beginning of this section the evaluation, criteria defining good results were set. Throughout the chapter, we looked at experimental results indicating that ERGo is able to handle more interrupts keeping agents longer alive and Figure 6-9 demonstrates that the approach scales well without sudden quality fall-offs. ERGo is only in the case of grooming behind the Flexible Latching as this would have forced a specific signal for "enough" grooming, which did not exist in Flexible Latching. However, ERGo's integration requires less hand-tuning and ERGo itself is well encapsulated and more robust, based on its independent internal ramp and the usage of asynchronous signals. The next section concludes the experimental results and presents potential areas for further improvement.

For upcoming experiments it would be interesting to have a closer analysis of the impact of noise and interruptions on ERGo augmented behaviour and verify the preliminary findings. A focus of future research could also be on identifying the effects of the inclination gain and the exponential modifier for the gain on the robustness of generic agents.

6.6 Concluding An Augmentation for Behaviour Arbitration

The current results indicate that the ramp function outperforms the Flexible Latching in certain scenarios even when it is not hand-adapted for the particular problem. It offers some unique features which offer additional potential. The ramp acts only upon a small set of signals and needs less fine-tuning to perform well. It has an easy-to-understand visual representation of the maintenance and inhibition process, presenting what can potentially be called a novice-friendly or intuitive approach to alter the arbitration process. Future work is needed on the influence of finer grained prioritisation and the control of the ramps precise inclination gain for distinct behaviours. This analysis could provide valuable insights and show the extent to which the ramp can scale to a variety of problems.

More generally, the ability of lightweight cognitive architectures and their importance in a variety of domains has been discussed. Based on the lower computational overhead and high practical applicability, it seems reasonable to give more attention to research on advancing those architectures to fully explore more restrictive environments. This investigation of light-weight augmentations could provide fruitful results, for example for the games industry, by expanding beyond the current capabilities of agent design and architectures, but also to robotics, and scientific and philosophical simulations.

To create a better understanding of challenges in those domains, a development project was started to apply the developed light-weight augmentation to digital games, aiming at experiments with perceived behaviour selection and how different users compare selection processes in virtual agents. For this, a first prototypical smartphone game for Android was developed—StealthierPOSH. The game integrated the results of Chapter 5 and the new light-weight behaviour augmentation presented in this chapter.

Architectures like those presented here, including the currently-popular spreading-activation architectures such as the Global Workspace Theory (GWT) [Baars, 2002; Shanahan, 2006] and LIDA [Franklin and Patterson Jr, 2006] cannot account for all of action selection [Rohlfshagen and Bryson, 2010]. Bryson [2005] argues that many details of action selection are handled by other, simpler neurological mechanisms in real primates, and computer science gives a good reason—combinatorics. Goal selection—the focus of interest—is a differentiable sub-part of action selection overall, one that requires competition between all of what may well be a limited set of contenders. The work presented in this chapter may provide a useful concept suitable for a large variety

of applications, including perhaps better understanding Nature itself.

In this chapter, we presented a novel approach—ERGo—for augmenting action selection system. The approach was compared against a similar approach in an agent-based modelling environment. The augmentation is a "white-box" plugin for existing systems and requires no initial adjustment to domain specific parameters and has a low computational overhead. ERGo is intended to handle highly competitive behaviours in noisy environments and introduces non-deterministic behaviour expression based on the effects of the summatory release mechanism. It was also included in a mobile game to demonstrate its low overhead and easy integration allowing developers to enhance their IVA without intensely reworking their arbitration mechanism or requiring extra computational resources. Due to its modular integration into POSH-SHARP as an optional package it increases the creative potential of POSH-SHARP without forcing developers to use it.

In the next section, an approach for evolving agents from human play data is presented. In contrast to similar approaches, the approach works from a clean slate, developing non-trivial agents from scratch in the form of executable Java programs. This approach can be used in a similar way to the ramp, discussed in this chapter can be used by designers to modify or create new agent behaviour supporting their creative expression. Additionally, the evolutionary approach uses human play data and, thus, creates models of the player.

Chapter 7

Evolutionary Mechanisms for Agent Design Support

In the previous chapter, an augmentation for action selection mechanisms was presented which allows agents to respond better to noisy environments or signals. This is especially important for Interactive Virtual Agent (IVA) in cases where the player is responding in ways which have not been considered at design time. The augmentation is intended for agents with multiple competing goals using only a low computational overhead on top of the existing system.

In this chapter, a novel approach for evolving agents is presented which uses genetic programming to create agents in the form of executable Java code. In contrast to other evolutionary or learning approaches such as artificial neural networks, discussed in Chapter 2.1.3, the agents derived in this work are in human-readable form which allow further modification and more importantly analysis of the agent. The approach is intended as a proof of concept showing the possibility of creating complex agents from human controller input in a "learning from demonstration" way.

7.1 Contribution

In this chapter, I demonstrate the application of Genetic Programming in combination with raw user input to the control of an IVA. The chapter is based on published work presented at the 17^{th} *Portuguese Conference on Artificial Intelligence* by Gaudl et al. [2015]. The chapter is the result of a collaboration with the University of California, Santa Cruz. The work integrates for the first time a similarity metric for strings into a learning game environment. The main contribution is the proof of concept for developing an evolutionary system which produces human-readable and modifiable

agent descriptions for games solely based on recorded human input. Thus, allowing the system to learn executable player representations by example. My contribution to this work is 80%.

7.2 Introduction

Designing intelligence is a sufficiently complex task that it can itself be aided by the proper application of AI techniques. In this chapter, a system that mines human behaviour to create automatically better Game AI is presented. Genetic programming (GP), described later, is utilised to generalise from and improve upon human gameplay. Moreover, the resulting representations are amenable to further authoring and development which is a central point of the entire work by following the direction to create a more robust development process through tool and methodological support.

When introducing the GP method for evolving IVA, the system uses unfiltered, recorded human play in the form of button input signals. The system uses the PLATFORMERSAI toolkit, detailed in Section 7.4, in combination with JAVA GENETIC ALGORITHM AND GENETIC PROGRAMMING PACKAGE (JGAP) as the evolutionary component. Meffert et al. [2000] developed JGAP as an evolutionary framework for GA and GP approaches. It is entirely written in JAVA and offers a mechanism to evolve fully working JAVA programs, in our case game agents. When the system is given a set of command genes (functions used by the agent), a fitness function, a genetic selector and an interface to the target application, it creates an initial set of programs which undergo an evolutionary process, altering the program pool for each generation. In the case of this case study, JGAP was extended to generate automatically artificial players by creating and evolving JAVA program code which is fed into the PLATFORMERSAI toolkit and evaluated using our player-based fitness function.

In the next section, we look at how the system derives from and improves upon the state of the art. Section 7.4 describes the system and its core components, including details of the fitness function. The chapter concludes by describing our initial results and possible future work.

7.3 Background & Related Work

In practice, making a good game is achieved by a good concept and long iterative cycles in refining mechanics and visuals, a process which is resource consuming. It requires a large number of human testers to evaluate and judge the qualities of a game. Thus, analysing tester feedback and incrementally adapting games to achieve better

play experience is tedious and time-consuming. This process is part of the general game development which is separated into the three to four subsequent phases discussed in Chapter 1.2.1. However, the feedback from testers which is needed to make adjustments is only available at certain times, and it requires rigorous preparation to get valuable feedback. Additionally, including a large number of testers increases the risk of leaking details about the game either onto the internet or to competitors, which is highly undesirable.

At that point, the approach presented in this chapter comes into play by aiming to minimise human-involved development time, manual adaptation and testing time. Nonetheless, the primary focus while optimising the existing industrial practice is to allow the developer to remain in full control of the process *and* the resulting agents.

7.3.1 Agent Design

Designing IVAs or game agents was initially very limited and involved only the rendering of clusters of pixels on the screen and the coordination of simple deterministic movement of those clusters. These simple early approaches were based on early hardware limitations; more sophisticated approaches were not feasible. With more powerful computers, it became possible to integrate more advanced approaches. The original game PAC-MAN contains a very restricted logic system for controlling the game characters (ghosts) but this concise system allows the developer to understand and grasp most of the code. Nonetheless, it was still complex enough that during the development of the version for the Atari system, new errors were introduced into the logic as described by Montfort and Bogost [2009]. Current games make the design of the contained agents more complex and time-consuming, a negative factor in the development process discussed by Mateas and Stern [2003] and detailed in Chapter 2.2.1. To address this issue, more sophisticated approaches such as BehaviorTree (Bt) were introduced when designing agents.

In 2002, Isla introduced the BehaviorTree for the game Halo, later elaborated by Champandard [2003]. Bt has become the dominant approach in the industry as it allows for a very visual structure of agents. The approach developed on Chapter 5.3 is functionally very similar and also can be visualised and edited in a structurally simple way. Even though the contained plan in Parallel-Rooted Ordered Slip-Stack Hierarchical (POSH) is a directed graph, the tree visualisation and simplification the editor provides are also favoured by the industry, apparent in the existence of the described editors in Chapter 2.3. Those approaches are designed to allow game designers and programmers to develop complex agents. For sophisticated or large agents, a tree design still contains large nested trees which require manual adjustment during development

and design. As the game evolves from an early prototype allowing a character to navigate between spaces, new features need to be added to convey the intended behaviour. A way to support or replace this iterative adjustment and development of agents is using generative approaches.

7.3.2 Generative Approaches

To generatively develop agents, techniques from artificial neural networks or genetic algorithms are mostly used, see Chapter 2.1.3. Holmgard et al. [2014]; Ortega et al. [2013] build models to create better and more appealing agents using generative systems. In turn, the resulting generative agents use machine learning techniques to increase their capabilities. Using data derived from human interaction with a game—referred to as human play traces—can allow the game to act on or *re-act* to input created by the player. By training on such data, it is possible to derive models able to mimic certain characteristics of players. One obvious disadvantage of this approach is that the generated model only learns from the behaviour exhibited in the data provided to it. Thus, novel behaviours are not accessible because a player never exhibited them. Orkin [2005] describes the emergence of new behaviour as one of the benefits of their approach, but the novelty of the result is very limited.

In contrast to other generative agent approaches [Perez et al., 2011; Togelius et al., 2012; Ortega et al., 2013], the presented work combines features which allow the generation and development of truly novel agents in a directed manner. The first is the use of un-authored, recorded player input as direct input into our fitness function, this allows the specification of agents only by playing. The second feature is that the agents are actual programs in the form of Java code which can be altered and modified after evolving into a desired state. This possibility of amending a learning agent creates a white box solution. While Stanley and Miikkulainen [2002] use artificial neural networks (ANNs) to create better IVA and enhance games using Neuroevolution, the developed system utilises genetic programming [Poli et al., 2008] for the creation and evolution of artificial players in human readable and modifiable form. The most comparable approach is that of Perez et al. [2011] which uses grammar based evolution to derive BTs from an initial set and arrangement of sub-trees. Using an initial set of agents speeds up and directs evolution in a way which reduces the possible space of solutions. In contrast to their approach, the evolutionary method introduced in this chapter starts with a clean slate to evolve novel agents from scratch.

To better replicate human behaviour or to model human-plausible behaviour, it is essential to understand how humans express themselves. Our focus in this chapter is not on understanding the complete reasoning process behind certain expressed

behaviours. Systems such as ACT-R [Anderson, 1993], SOAR [Laird et al., 1987] or ICARUS [Langley et al., 1991] are built for that purpose and require a tremendous amount of initial design and computational power to run agents. We aim to purely abstract a thin and less complex layer on top a much simpler model. In our particular case, the work tries to replicate human gameplay behaviour exhibited in digital games. An assumption of the work is that using raw input data to create "white-box" agents is a useful first step and a good starting point to be able to replicate a particular expressed behaviour given a finite, small set of information about the situation and environment the expressed behaviour originated in. We use the term superficial to differentiate the approach from the previously mentioned heavy-weight approaches such as ACT-R and focus on a more lightweight model of human expressed behaviour modelling.

7.3.3 Genetic Programming

Evolutionary Algorithms (EA) extend the metaphor of biological evolution into the domain of computation systems [Bäck, 1996; Schwefel, 1993]. They adopt concepts from genetics such as genes, mutation, or recombination to describe a process of adjusting programs or program parameters in an incremental process. The field of EA can be subdivided into four areas discussed in Chapter 2.1.3. Here, Genetic programming is used in an approach to evolve artificial agents. GP was first introduced by Koza [1992] as an extension of genetic algorithms.

For explaining the method we can use following terminology: A given GP uses I as the search space containing all individuals $a \in I$ and $F : I \rightarrow \mathbb{R}$ as the fitness function which assigns a real-valued fitness value to each a. The size of a population of programs, in our case agents, is specified by μ for the parent population and λ for the offspring population size. $P(t)$ represents a given population at time t and consists of individuals of type a. To alter a population mutation, recombination and selection operators are utilised, each with specific characteristics, Θ.

The GP uses following steps to evolve programs based upon the biological concepts of evolution. Before starting the evolutionary process, the time $t = 0$ is set to track the evolution of the population $P(t)$. The population $P(t = 0)$ is initialised with either random or predefined individuals $P(t = 0) = \{a_0(t = 0), \ldots, a_\mu(t = 0)\}$. The initialisation is a crucial step for each EA and is different for each actual approach. Depending on the modification criteria Θ_r, Θ_m, an initial evaluation of the pool is carried out to assign each individual a a fitness value. For a genetic programming approach, Θ_r is the most important parameter and is dominating the other operators because recombination is the driving force behind GP. The mutation operator—Θ_m—is in GP only used to as a secondary operator to introduce noise and a small random

effect into the process, Bäck [1996].

Genetic Programming Process:

1. (Recombination)
   A new population is created $P'(t) = recombine(P(t), \Theta_r)$ using the parent generation and the specific criteria for creating a new population $P'(t)$. The criteria for the current work are given in Figure 7.1, where the parent percentage is specified as well as the recombination pool size, κ as well as the recombination method.

2. (Mutation)
   To better explore the solution space and to reduce the chances of getting stuck in a local optima, $P''(t)$ is created by taking each individual $a'(t) \in P'(t)$ and exposing it to the possibility of mutation $a''(t) = mutate(a'(t), \Theta_m)$ putting the exposed individual into $P''(t) = \{a''_0, \ldots, a''_\lambda(t)\}$. For GP, the mutation probability is extremely small, in some approaches mutation is not used at all.

3. (Evaluation)
   During this step, the population is evaluated by calculating the fitness of each individual, taking the total number of offspring into account. $F(t) = evaluate(P''(t), \lambda)$

4. (Selection)
   During selection a set of offspring is chosen to go into the new generation as λ can be larger than μ. The new population $P(t+1) = selection(P''(t), \Theta_s)$ is created from $P''(t)$ and additional criteria are applied Θ_s according to the used approach.

5. (Clean Up and Increment)
   This step is important in computational systems especially when performance and memory are restricted. It offers a way to optimise the speed of the approach but due to the different focus of the current case study is not further explored here. All elements which have not made it into the previous generation are cleared, and approach dependent measures are taken either to save the fittest individuals or insert specific individuals into the new generation by force. Additionally, the generation counter is incremented.

Most GP approaches use lisp-like structures or decision trees to describe the programs a_i. Based on the re-combination method, those representations are then altered by taking a number of parent programs and creating a new child. The main difference to other evolutionary approaches is that the size of a program $a_i(t) \in P(t)$ is not fixed

but can change over time allowing a greater exploration of the solution space. Once the last step in the GP process is reached, the loop starts again until a solution criterion is reached and the approach either returns the best solution program or a list of the progressions of all potentially best programs.

7.4 Setting and Environment

Evolutionary algorithms have the potential to solve problems in vast search spaces, especially if the problems require multi-parameter optimisation [Schwefel, 1993, p.2]. For those problems, humans are typically outperformed by programs according to Smit and Eiben [2009].

Using genetic algorithms (GA) allows a system to exploit a given program by evolving the parameters to better fit the solution. As a base, an already given program is used, which is not changed over the course of the evolution. This also means that the resulting genotype has a fixed length of chromosomes. Genetic programming (GP), as discussed earlier, explores the solution space more broadly by altering the structure and size of a program over time. This broader exploration is ideal for identifying entirely unknown solutions as the genotype has no fixed length. The size of the solution program tree can vary considerably making it possible to evolve programs describing more sophisticated approaches to a problem. The downside of using any evolutionary method is that finding a solution takes time and evolving programs using GP takes more time compared to pure parameter optimisation because the approach is exploring the solution space with a less strict focus.

JGAP uses a pool of program chromosomes P and evolves those in the form of decision trees (DTs), see Figure 7-3. For our experiments the PLATFORMERSAI toolkit (`http://www.platformersai.com`) is integrated with JGAP. It consists of a 2D platformer game (see Figure 7-1). It is similar to existing commercial products and contains modules for recording a player, controlling agents and modifying the environment and rules of the game.

The *Problem Space* is defined by all actions an agent can perform. Within the game, agent A has to solve the complex task of selecting the appropriate action each given frame. The game consists of A traversing a level which is not fully observable. A level is 256 spatial units long, and A should traverse it left to right. Each level contains objects which act in a deterministic way. Some of those objects can alter the player's score, e.g. coins. Those bonus objects present a secondary objective. The goal of the game, moving from start to finish, is augmented with the objective of gaining points. A can get points by collecting objects or jumping onto enemies. In Figure 7-1,

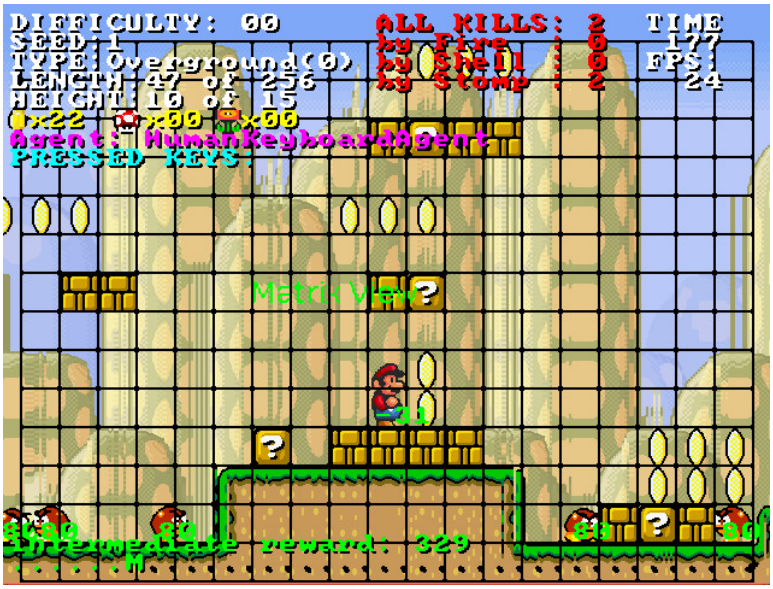


Figure 7-1: The PLATFORMERSAI toolkit allows researchers to develop and research agents for real-time two dimensional platform games. The player is at the centre of the figure surrounded by a sensory grid overlay. The grid illustrates the agents capability in sensing the world. The game is resembling the SUPERMARIO game series developed by Nintendo.

five enemies are visible in the lower half, as well as multiple coins and the player at the centre of the 20×20 grid. To make it comparable to the experience of similar commercial products, a realistic time frame is used similarly to the one a human would need to solve a level, 200 time units. The level observability is limited to a 6×6 grid centred around the player, cf. Perez et al. [2011].

Agent Control is handled through a 6-bit vector C: $left$, $right$, up, $down$, $jump$ and $shoot|run$. The vector is required each frame, simulating an input device. However, some actions span more than one frame. This is a simple task for a human but quite complex to learn for an artificial agent. One such example, the high jump, requires the player to press the jump button for multiple frames. Our system has a gene for each element of C plus 14 additional genes formed of five gene types: sensory information about the level or agent, executable actions, logical operators, numbers and structural genes. All those are combined on creation time into a chromosome represented as a DT using the grammar underlying the JAVA language. Structural genes allow the execution of n genes in a fixed sequence, reducing the combinatorial freedom provided by JAVA.

Evaluation of Fitness in our system is done using the Gamalyzer-based play trace metric which determines the fitness of individual chromosomes based on human traces as an evaluation criterion.

For finding optimal solutions to a problem, statistical fitness functions offer near-optimal results when optimality can be defined. For this work the main interest lies in understanding and modelling human-like or human-believable behaviour in games to aid the design of DEEPER AGENT BEHAVIOUR IVAs. There is no known algorithm for measuring how human-like behaviour is; identifying this may even be computationally intractable. A near-best solution for the problem space of finding the optimal way through a level was given by Togelius et al. [2010] using the A^* algorithm. This approach produces agents that are exceptionally good at winning the level within a minimum amount of time but at the same time are clearly distinguishable from actual human players. For games and game designers, a less distinguishable approach is normally more appealing—based on our initial assumptions.

7.5 Fitness Function

Based on the biological concept of selection, all evolutionary systems require some form of judgement about the quality of a specific individual—the fitness value of the entity. Our *Player Based Fitness* (PBF) uses multiple traces of human, t_h, and agent, t_a, players to derive a fitness value by judging their similarity. For that purpose, we integrate the Gamalyzer Metric—a game independent measurement of the difference between two play traces. It is based on the syntactic edit distance d_{dis} between pairs of sequences of player inputs introduced by Osborn and Mateas [2014]. It takes pairs of sequences of events gathered during gameplay along with designer-provided rules for comparing individual events and yields a numerical value in $[0, 1]$. Identical traces have distance $d_{dis} = 0$ and incomparably different traces $d_{dis} = 1$.

Gamalyzer finds the least expensive way to turn one play trace into another by repeatedly deleting an event from the first trace, inserting an event of the second trace into the first trace, or changing an event of the first trace into an event of the second trace. The game designer or analyst must also provide a comparison function which describes the difficulty of changing one event into another.

The other important feature of Gamalyzer, warp window ω, is a constraint that prevents early parts of the first trace from comparing against late parts of the second. This is important for correctness (a running leap at the beginning of the level has a very different connotation from a running leap at the pole at the end of each stage).

For our purpose, only the input commands players use to control the agent are encoded—the six commands introduced earlier. This allows us to compare against direct controller input for future studies and to help designers sitting in front of the controls analysing the resulting character program. The PBFcurrently offers two pa-

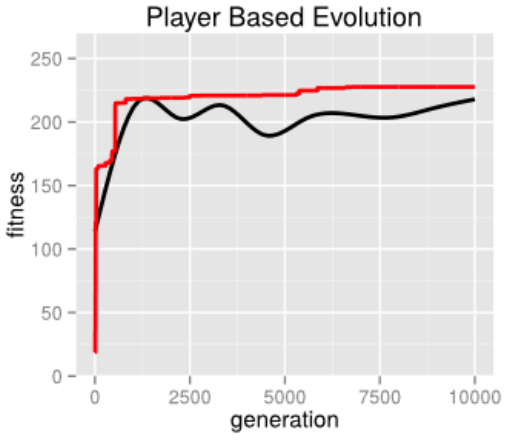


Figure 7-2: The evolved agents' fitness using PBF (10000 generations), in red the fittest individuals, in black the averaged fitness of all agents per generation.

Table 7.1: GP parameters used in within the presented work.

Parameter	Value
Initial Population Size	100
Selection	Weighted Roulette Wheel
Genetic Operators	Branch Typing CrossOver and Single Point Mutation
Initial Operator probabilities	0.6 crossover, 0.2 new chromosomes, 0.01 mutation, fixed
Survival	Elitism
Function Set	$ifelse$, not, &&, ||, sub, $IsCoinAt$, $IsEnemyAt$, $IsBreakAbleAt$, ...
Terminal Set	Integers [-6,6], $\leftarrow$, $\rightarrow$, $\downarrow$, $IsTall$, $Jump$, $Shoot$, Run $Wait$, $CanJump$, $CanShoot$, ...

rameters: the chunk size, cpf, and the warp window size, ω. The main advantage over a pure statistical fitness function is that a designer can feed our system specific play traces of human players without having to modify implicit values of a fitness score.

To make a stronger emphasis on playing the game well, we create a multiobjective problem using an aggregation function g to take Δd—the moved distance—and the fitness f_{ptm} for an agent using the player-based metric PBF into account, see formula 7.1. Using g we were able to put equal focus on the trace metric, $f_{ptm} \in [0 \dots 1] \subset \mathbb{R}$, and the advancement along the game, $\Delta d \in [0 \dots 256] \subset \mathbb{N}$.

$$f(a) = g(f_{ptm}(t_a, t_h), \Delta d) \tag{7.1}$$

7.6 Results & Future Work using GP

Using the experimental configuration and the PBF fitness function makes it possible to execute, evaluate and compare PLATFORMERSAI agents against human traces. For the parameters needed to define the approach, the settings from table 7.1 are used. As a selection mechanism, the weighted roulette wheel is used and additionally the fittest individual of a generation is preserved. For the recombination, a single point tree branch crossover on two selected parent chromosomes is used, and the resulting child is after recombination exposed to a single point mutation before it is put into the new generation.

Figure 7-2 illustrates the convergence of the program pool against the global optimum. Good solutions such as the agent in Figure 7-3 are on average reached after 700 generations when an agent finishes the given level. First experiments show that the approach indeed is able to train on and converge against raw human play traces without stopping at local optima, visible in the two dents of the averaged fitness (black) diverging from the fittest individual (red).

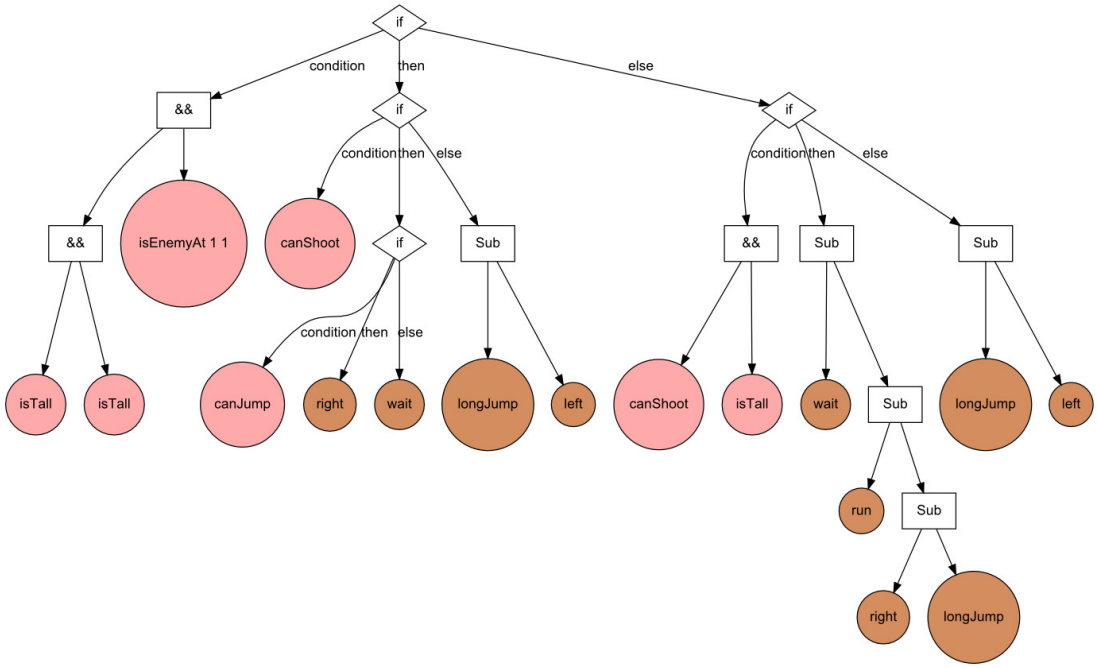


Figure 7-3: An evolved agent that is able to pass the level from start to finish. The agent emerged after 700 generations and is the result of using the human play trace in combination the PBF metric.

A next step would be to investigate the generated modifiable programs further and analyse their benefit in understanding players better. However, the current solution already offers a way to design agents for a game by simply playing it and creating learning agents from those traces. Other possible directions could be the expansion of the model underlying Gamalyzer to model specific events within the game rather than pure input actions. This should provide interesting feedback and offer a better matching of expressed player behaviour and model generation.

The current agent model consists of an un-weighted tree representation containing program genes. Currently, subtrees are not taken into consideration when calculating the fitness of an individual which is common in GP as sub-tree evaluation requires extra computing time and a specialised approach. By including those weights it would be possible to narrow down the search space of good solutions for game characters

dramatically, also potentially reducing the bloat of the DT. So, to enhance the quality of our reproduction component, it might be a possibility to investigate the applicability of behavior-programming for GP (BPGP) Krawiec and O'Reilly [2014] into the system.

7.7 Evolutionary Mechanisms Summary

In this chapter, we examined and discussed a proof of concept for creating non-trivial agents from unfiltered human play data. The approach evolved agents which will able to complete a game and reproduced similar results to the "original human" player. Extending this concept, designers are now able to express a certain behaviour in a game environment while playing and use the presented genetic programming approach to evolving artificial players, resembling this behaviour. The resulting behaviour can, later on, be further modified and tuned by programmers. However, designers and novice users are now able to create first prototypes of intentional agent behaviour without the need to touch the underlying programming environment. This approach not only supports the design of robust agents but also the separation of tasks between programmers and designers.

A further benefit of this approach is to analyse the resulting agent representation, not with the focus of creating better agents but to understand the user's motivation and approaches to the game better. Using such an automated approach based on observation allows additional insights into possible player models and provide a completely new direction of future research.

In the next chapter, the results of this thesis are put into relation with the current state of the art discussed at the beginning of this work. Different aspects of IVA design are discussed to show research opportunities which became visible during this thesis but could not be addressed due to the focus on creating a robust, simplified process for agent development. In the next chapter we compare the literature to the contributions of this thesis and discuss specific points of the new approaches in wider detail which opens up new possible research directions for future work.

Chapter 8

Discussion & Future Work

In the previous chapters, we presented the individual contributions of this thesis in isolation, addressing different parts of our research question. When we now take a step back and remember the initial question which motivated this work,

How can the design cost for behaviour-based AI be reduced?

we are able to view the contributions as partial answers that form one answer to our question. We introduced and discussed:

- *Agile Behaviour Design* as a new methodology for game AI development by combining agile elements of software processes and an existing agent design approach. techniques a more directed and supporting approach.The new methodology presents guidelines for developing agents reducing the interdependencies between team members and strengthening their independence. It also supports the development of games in industrial contexts by incorporating features from SCRUM—an agile process model.

- The POSH-SHARP agent framework for creating agents allows the development of light-weight game AI systems and includes extensions such *behaviour versioning* and *behaviour inspection* to increase the robustness of new agent systems and support the development of game AI for novice users. The approach was developed with deployablity and mobile device support. The systems addresses a subset of the identified weaknesses of all surveyed architectures and provides a novel platform for experimentation.

- A survey of the state of the art of game AI techniques was created as a primer for future Interactive Virtual Agent (IVA) research and used to identify the potential points for advancing or creating a new approach. This survey integrates academic

and industrial research on components, architectures and approaches to IVA into a wholistic view on the topic.

- To enrich the pool of agent capabilities, ERGo, a low-cost mechanism for altering the selection process of goals and behaviours, was introduced into POSH-SHARP. This novel augmentation is also applicable to other systems due to the low coupling and its "white-box" nature, requiring no domain specific parameters. It introduces a new form of memory into the selection process, the extended ramp goal. This summatory memory allows a behaviour to take control using an internal motivational state or need. ERGo is a light-weight bio-mimetic mechanism for creating non-deterministic behaviour that can be designed. It addresses the industrial need for easy to integrate but flexible approaches to goal selection.

- A new GP system using recorded input data from human players is presented to offer the possibility to evolve artificial players in a form amendable for further authoring. The resulting agent design approach allows for a "learning by example"-way of evolving non-trivial, understandable and amendable agents as executable program code. The developed approach evolves artificial players using PLAY TRACEs in the form of Java program code allows for the inspection of possible underlying models of the player's motivation or reasoning process.

To relate those contributions back into current game development, let us have a look at the current development of games again. Game development is a time-consuming process and current games require a vast amount of resources during their production, similar to movie productions. A state of the art game such as GRAND THEFT AUTO V by *Rockstar Games* required a development budget of over 250 million US Dollars and took five years of development for a team of approximately 1000 people. During those five years, the same stages of game development as described in Chapter 1.2.1 were followed, leading through a staged process of development from concept to post-production. Even for smaller productions, the process of development is similar.

To allow the development of such diverse projects, process models from software development are currently used to guide the creative process as well as the overarching development of a game. According to Keith [2010], SCRUM is the most suited for digital game development as it combines the flexible and iterative nature of agile processes with a guided model to integrated the three stages of software development for games. A problem in traditional software process models including the one presented by Keith [2010] is the integration of creative input and design which is needed for game AI development and more specifically for agent design as artificial agents or IVAs represent the most complex parts of games in terms of their logical design.

O'Donnell [2012] argues that a new or adapted process is needed to better facilitate the creative nature of the game development approach and that "just" a software development approach is not enough. Based on the process analysis presented in Chapter 3, essential process steps of IVA architectures and their design approaches were analysed and the System-Specific Step was developed to capture and describe the individual differences between the three, Abl, **F**earNot! **A**ffec**ti**ve **M**ind **A**rchitecture (FAtiMA) and Behaviour-Oriented Design (Bod). Two of the approaches,Abl and FAtiMA, do not provide an explicit approach to design or development. However, both approaches are used by close-knit communities which iteratively learn from more senior team members. Thus, they employ an implicit development process that is not explicitly communicated but implicitly inherited. By analysing the development teams, essential steps for both systems were identified by Grow et al. [2014]. Related weak points such as missing debug support, missing architectural design knowledge and the need for visualisation of the executing behaviours became more transparent for each approach. Grow [2015] extends those findings and proposes additional components to support the development of Abl IVAs in her advancement proposal. Weber et al. [2011] includes his own design into the Abl system and introduces managers which alter the overall design of an Abl behaviour tree by including parallel behaviours for each manager. This makes the usage hard for novice developers or more design focused team members as no coherent, explicit concept is present. Based on the literature survey and the conduced informal interviews common elements between most IVA are existent which allowed us to propose the System-Specific Step (sss) as a way of identifying the underlying implicit design rules which are otherwise hidden.

The same methodology can be applied to BehaviorTree, which is a data structure to design complex dynamic systems. Isla [2005] and Hecker [2009] discuss their usage and development approach and the difficulties that they encountered when developing robust, efficient game AI systems. However, they focus their attention on programmers and omit most of the control for non-programmers reducing their creative expression to the selection of navigation points in the virtual environment where events should occur. Anguelov [2014] describes in his argument for a programmer-driven approach to BehaviorTree (Bt) the need for more expressive tools and a system usable by designers, as argued for by O'Donnell [2012].

The posh-sharp system we propose is structurally similar to Bt, based on the underlying structure Parallel-Rooted Ordered Slip-Stack Hierarchical (POSH), and provides a guided approach to agent design which is missing from Bt. Guidance is suggested as one of the elements identified using the sss and in the survey which aids the design and similar to simplicity of the approaches supports industrial applicabil-

ity. Behaviour-Oriented Design, introduced by Bryson and Stein [2001] for creating reactive BBAI, comes with an explicit development methodology that focuses on fast prototyping and iterative development. However, the original BOD was designed for the development of behaviour-based robotics and agent simulations and is mostly used in academic contexts for developing light-weight agents. BOD is not adjusted for the requirements of distributed or shared development and time-dependent teamwork. Integrating the knowledge from Chapter 3 and features from SCRUM into an extended version of BOD was tested in two projects presented in Chapter 4. The first is a STARCRAFT case study and its extended version using a guided design to arrive at a sophisticated agent. This process further separated the activities of design and implementation and guided the scheduling of development based on an overarching process instead of pure iterative development. The second is the development of a mobile game integrating the extended ramp model from Chapter 6. Applying the same methodology to the development across platforms was less problematic as the underlying structure—POSH-SHARP—supports a platform-independent approach. However, taking the more restricted resources available on a mobile device into account required changes to the feature board and revisiting existing features whenever performance was impacted. Nonetheless, the fully layouted behaviour library required less interaction between planning/design layer and underlying behaviour layer than incremental development.

The original BOD methodology starts with a working version of a simple plan for an agent expanding it over iterations; the process is described by Partington and Bryson [2005], illustrating the initially simple plan and incrementally adding more behaviour primitives and plan elements. To allow for shared work on a project, the interdependence between its components should be as minimal as possible, thus, revisiting the underlying layer of a project introduces additional changes in all layers above. The iterative nature of the original BOD introduces this frequent revisiting and the alteration of elements on all levels, a process which works well only for small teams; larger projects or projects which introduce dependencies between components and specific team members require an altered approach. Additionally, SCRUM introduces a visual component, the feature board discussed in Chapter 5, which allows the developer to emphasise the importance of different development steps. By augmenting BOD and creating an AGILE BEHAVIOUR DESIGN, it is possible to reduce the dependence of team members and guide the design better towards "on-time" development of the game AI system. The discussed methodology would also, due to the discussed similarity, be applicable to BT to create an integrated agile process for game development with their currently favoured approach. ABL and FATIMA use implicit development processes

which require a novice author to extract knowledge from more senior system developers. This knowledge transfer hinders the spread of ABL and FATIMA into other communities. Chapter 3 presented similarities shared between all three IVA systems, thus, it would be possible to create a new version of BOD or AGILE BEHAVIOUR DESIGN to guide the development in those systems reducing the peer-to-peer knowledge transfer and increasing the overall understanding of the system design.

After an adjusted methodology and guideline were proposed and tested, which provide a flexible but more game-focused approach for agent development, the identified issues with the underlying framework were addressed in this work. Weber et al. [2010a] provides two extensions to ABL in the form of a prototypical debugger and behaviour tree visualiser. Those extensions align with the feedback gathered by Grow et al. [2014] in regards to more advanced development support. The integration, stability and configuration of both extensions seems to be too complex or problematic because they are not used in later projects, an observation that is supported by the interviewed ABL authors. This stresses the need not simply for functionality but also for a certain degree of maturity and robustness of new extensions to be useful.

The complexity of employing an existing approach or system also seems to reflect on the usage of more fully cognitive agent architectures such as SOAR, ACT-R or ICARUS. The structure of all three systems is highly similar and the underlying concept of using production rules to encode knowledge is shared because all three extend the idea of a unified theory of cognition introduced by Newell [1994]. SOAR and ACT-R have been actively maintained for over two decades, making them the oldest still actively developed fully-cognitive systems. Most developed and proposed systems which are discussed in research on cognitive architectures are not maintained or even available anymore. This inaccessibility and deprecation of systems make actual comparisons of benefits of the underlying concepts nearly impossible. ICARUS, a relatively recent system, is officially still maintained but not freely available, similar to MIT's *cX* system, whereas both SOAR and ACT-R provide current versions.

The usage of all discussed academic platforms is, nonetheless, limited to narrow encapsulated communities. Reasons could be that the design of sophisticated agents in those architectures requires a large amount of computational resources as illustrated by Wintermute et al. [2007]. Wintermute et al. are only able to utilise SOAR for the high-level selection of goals. Similar issues emerge with the discussed cX system which requires a computationally powerful system for controlling six agents, discussed in Chapter 2.2.7. Comparing those architectures and extracting common elements creates an interesting perspective on them and allows the augmentation of an existing system to move towards the more cognitive systems. All three systems share to some

extent a modular structure, or at least, they can be abstracted into one. The usage of production rules makes it possible for authors to specify procedural knowledge which can be used by the system. All three systems also alter and modify their procedural knowledge over time to model learning. This is either done as part of the update cycle happening roughly every 50ms or whenever a particular block of procedural knowledge is used. The usage of declarative knowledge in the form of statements is also common in all three systems. The usage of other forms of declarative knowledge, such as images or audio, was introduced into SOAR after it had been a part of the ACT-R system and ICARUS uses memory snapshots to represent any form of declarative knowledge. The usage and updating of stored knowledge in declarative or procedural form make those systems quite powerful but also requires a large amount of computational resources. This requirement for computational resources renders all of the three fully cognitive systems nearly unusable for the usage of complex agents when resources are limited. Knowledge is used either for short-term memory of for long-term memory and in all three system transitions from the first to the second with regular usage.

Games rarely need access to and updating of long-term memory though it would potentially support DEEPER AGENT BEHAVIOUR behaviour. However, the illusion of long-term memory in the form of a given set of long-term goals suffices most of the time and game interactions between a player and the same IVA seldom last for longer periods of time. Removing the cyclic process of retrieving and updating long-term memory in the form of large amounts of production rules would make architectures more light-weight and reduces a large amount of computation. ABL and GOAP are examples for that. The planner combines and integrates goals which are structured by a designer but the retrieval process is handled by the system. The *F.E.A.R.* system discussed by Champandard [2007b] uses GOAP but the system does not integrate long-term memory as it goes beyond the scope of agents which only exist for short amounts of time. In contrast to this, ABL uses an object oriented memory system (WoMe) resembled by the *cX* of Burke et al. [2001]. The WoMes can be used for long-term memory but only present declarative knowledge, procedural knowledge is not captured by the system as the system does not include an existing approach for persistent memory between sessions.

Orkin [2005] uses a blackboard to integrate short-term memory into his GOAP system in a similar fashion to Mateas and Stern [2003] with ABL in FAÇADE. Checking and integrating memory introduces complexity that requires planning, both in the designer and on the system side. POSH-SHARP provides two forms of memory inspired by the cognitive architecture and models of the animal brain. The first one is the BEHAVIOUR BRIDGE which provides a central channel for sharing knowledge and re-

ducing the computational overhead of the agent. This concept of a loose coupling and a central approach information exchange is similar to a blackboard but follows the idea of a *Listener Pattern* instead of a storage space. To support a strong separation of behaviours and to tackle the complexity of connected states, game developers moved from Finite-State Machine s (FSMs) to BT. POSH-SHARP extends this idea by having an independent pool of behaviours for an agent and similar to the *corpus callosum* in the mammalian brain a central route for sharing and exchanging information between components. The second mechanism is similar to how ICARUS and ACT-R handle the activation of productions.

POSH-SHARP integrates ERGO—the extended ramp goal model. ERGO is inspired by goal cells in the *basal ganglia* and the ramp-like activation process during goal pursuit described by Redish [2012]. In contrast to more complex models such as modular utility models or ICARUS's production selection process, ERGO augmented behaviours can be individually selected but require no initial adjustment. An augmented behaviour accumulates activation until it is able to execute its goal. Using this model a state-like transition is added to the selection process of a goal which is based on a consistent model but allows for dynamic non-deterministic behaviour to be designed. The motivation behind the design of a general model for behaviour augmentation was to present "white-box" modules which can aid the design flow. This is based on the idea underlying the decorators described by Champandard and Dunstan [2013]; Isla [2005]; Anguelov [2014]. Decorators modify the connected tree node and change or augment its result. Nonetheless, Complex decorators containing state can completely obscure the DECISION-MAKING SYSTEM (DMS). Thus, a designed augmentation based on the animal-like pursuit of goals was developed to motivate a more dynamic action selection.

The AGILE BEHAVIOUR DESIGN offers a more directed process and providing a memory system to allow for more sophisticated IVAs has been discussed as beneficial to support more cognitive models. Most Integrated Development Environments (IDEs) further support a programmer by allowing code completion but using a large behaviour library require additional considerations. Version control was introduced into software design to track changes during the development but those changes operate on a file level. POSH-SHARP introduces *behaviour versioning* as a way to manage individual behaviour primitives; this process increases the robustness of a plan by offering the designer the opportunity to go back to an older version of a primitive or compare how both primitives work and behave, on a design level. In contrast to traditional refactoring of methods, *behaviour versioning* keeps the original behaviour primitive and allows the programmer to add a second version under a new method name but referring to the same primitive identifier. The feature extends the principle of traditional software

version management to a behaviour level on which designers or novice developers are working and allows them to benefit from a flexible and robust mechanism to compare different implementations without the need to consult a programmer. The *automatic behaviour loading* provided by POSH-SHARP is the second mechanism which reduces touching the underlying code. This process is similar to how ABL chains behaviours or POGUMUT handles extra attributes used for inspection but on a more abstract level. A designer describes elements and later on a programmer implements the element inside a behaviour class. However, the programmer is able to adjust the underlying code by switching and versioning primitives using POSH annotations. This is completely hidden from the designer offering a much cleaner interface from the underlying library.

8.1 Future Work

In this thesis, multiple projects were worked on and discussed to create an integrated approach to more robust agent design and there exists a variety of different paths which are left untouched or which could be extended. In this section of few of them will be named and explained based on their merit to the proposed approach to agent design or IVA development in general.

The new AGILE BEHAVIOUR DESIGN process extends BOD by integrating features from other agile processes such as SCRUM. The approach has been used in two projects and has been presented to developers from different systems such as ABL and FATIMA a more detailed analysis on novice developers is needed to identify further areas of improvement both for the application to industry as well as for the application in other scientific communities. This evaluation can also provide insights for systems such as SOAR, ACT-R or ABL as they are only applied in close-knit communities and their impact on a broader audience.

The application of a light-weight cognitive architecture to a complex dynamic problem such as STARCRAFT was covered in Chapter 4.3 and an evaluation of the prototypical but shallow strategy was conducted. As a next step, an analysis of the performance and design of the extended strategy would create a more measurable comparison against other state of the art approaches to DMS for REAL-TIME STRATEGY (RTS) games and allow for the identification of potential points of further improvement.

The extended ramp, introduced in Chapter 6, is a light-weight, general-purpose augmentation for action selection mechanisms. Future work could involve a detailed analysis of the underlying inclination gain based on initial priorities of the behaviours. This analysis would allow a more fine-grained approach to scheduling the arbitration process than currently available. To support the claim of general applicability, creating

case studies and including ERGo in different game environments would be advantageous for evaluating the impact of modules which are more generalisable. Analysing the applicability in different environments would also support the claim for its robustness and give examples of its integration, making it easier for professionals to transfer the approach to different game development tools once it conforms to an industrial environment. A first step was taken with the inclusion into the StealthierPOSH game but a user evaluation of the aimed effect of the system could provide insights into the understanding and communication of less deterministic decision processes.

The Behaviour Oriented Design methodology in combination with posh-sharp itself categorises as a framework for providing a cognitive architecture to implement cognitive agents. As argued in Chapter 7, which describes a process for evolving game playing agents for a 2D platform game, the design of novel behaviour which expresses non-trivial behaviour through its actions is a complex task. By using genetic programming and human-provided input, it was possible to remove most of the tedious work on tuning and testing of agents. Additionally, the underlying grammar using Java genetic algorithm and genetic programming package (Jgap) and its decision tree representation is not restrictive enough for creating sophisticated cognitive agents within a reasonable time-span. JGAP provides a general purpose environment for evolving GP systems. However, it does not offer support for grammar-based GP which allows the inclusion of specialised rules. Additionally, the current fitness evaluator is unable to provide fine-grained feedback on the fitness values at a gene level. Next steps could involve the optimisation of the evolutionary process taking either the structure of the decision tree into account when evaluating the fitness by applying research from Krawiec and O'Reilly [2014] on behaviour programming for genetic programming or by speeding up the most time-consuming step in the process—the simulation of the agent within the environment. This last step is interesting because it would allow game developers to include evolutionary approaches into their system once the evaluation is fast and light-weight enough to run in the background of a game.

Evolving Bt has been shown to work with some success for small examples Lim et al. [2010] and the evolution of decision trees for more complex settings based on pure human feedback has been shown in Chapter 7. The evolutionary process which we discussed in Chapter 7 allows the development of amendable agents in human-readable form, while no programming skills are required. However, the inclusion of an evolutionary process into the a full agent framework such as posh-sharp would allow a more rigorous examination of its benefits and allow for a better transition into industrial practice.

Chapter 9

Conclusion

In this thesis, we addressed the question of reducing the design cost of human authors when creating behaviour-based AI for games. For that purpose we discussed approaches to game development, and more specifically game AI and IVA development. Based on the motivation to identify requirements for a more robust, creative process that better supports novice developers and designers, informal interviews were conducted. During those interviews potential points for advancements of current approaches were identified. The SSS is one result of approaching research question **RQ1**, where we examined the commonalities between agent frameworks and how to support the understanding of developing IVAs. SSS was discussed in Chapter 3 and presents a unifying element for most IVA architectures. It unites their analysis under a common schema should be cindered when building new agents or even new frameworks.

To investigate the requirements and circumstances for approaches to transition from academia to the games industry we examined the literature and current approaches (**RQ2**). The games industry tends to favour light-weight and flexible approaches which have a shallow learning curve, supported by existing step-wise instructions, showcases of implementations in games and vanilla textbook solutions. One recent case for a successful transition is Monte-Carlo TreeSearch (MCTS), discussed in Chapter 2.1.3, another possible candidate would be Bod. To pursue the question further, Bod was chosen as a base for improvement, aiming to suffice the previous requirements for a transition into industrial application. As a result, a new method—Agile Behaviour Design—was developed including features from a prominent industrial development model, Scrum. The new approach contains explicit guidance such as, which agent feature to work on next or how to initially structure the agent. This guidance can aid novice developers to ease the transition to a new approach and support robust development. The approach also addresses a fundamental issue existing in software-driven

game development, the strong dependence of authors on programmers when designing IVAs. The issue was addressed by stronger separating the tasks of designing the agent and implementing the underlying functionality.

To further support authors, a new agent framework based on POSH was developed, integrating the features that emerged during the informal interviews in Chapter 3. The new framework was applied to the development of a sophisticated StarCraft agent, demonstrating the Agile Behaviour Design approach in combination with POSH-SHARP. Due to the high similarity between POSH and in the prominent industrial approach BT, the design approach is ideally suited for BT as well, enabling game developers to use an explicit methodology when approaching game AI design. POSH-SHARP integrates most of the aspects of IVA development by borrowing concepts such as "goal design", a planning component, memory and the integration of perception from fully-cognitive architectures and other IVA architectures. The system employs further techniques from software development such as dynamic library inspection for *automatic behaviour loading* and *behaviour versioning* address robustness and ease of use questions formulated in Chapter 3. After adjusting the Behaviour-Oriented Design process, this new architecture for more robust game AI development was tested in the two projects, which have been mentioned earlier StealthierPOSH and the StarCraft agent, and was utilised while developing the foundations for an Unreal Tournament based coursework.

Based on what is employed in industrial developments and observable from the literature analysis, the games industry seems to favour less complex approaches, a claim which is supported by the industrial usage of BT and older but simpler approaches such as FSMs. This observation feeds into the results of research question **RQ2** and is also biologically plausible and in line with the BOD philosophy of Bryson [2000b] to try the simplest approach to get the job done first. It is also supported by BT's dominance over other more sophisticated approaches during talks at industrial conferences such as the Games Developer Conference (GDC) or the AiGameDev Conference. BT is a data structure and framework for implementing DMS but requires, similar to the previous approaches, guidance which can be given by Agile Behaviour Design. The approach has been applied to RTS game AI development and the development of a mobile game proving the flexible nature and multi-platform capabilities of POSH-SHARP.

The application of StarCraft build orders from user forums, demonstrated in Chapter 4.6, shows that it is possible to move the design of agents closer to specific target audiences and present an approach that does not require programming but still allows for the design of complex agents, addressing question **RQ4**.

The usage of genetic programming and human input to create non-trivial, amend-

able, artificial agents has been shown in Chapter 7. This evolutionary approach allows a designer to feed player data into a cyclic process which creates artificial players based on the input. The resulting process allows the development of agents in a "learning from example" fashion without the need for programming but compared to ANNs the resulting agent exists in a human-readable and amendable form which is suited for further adjustment and optimisation. This adds possible answers to questions **RQ3** to **RQ5**.

In this thesis, different aspects and approaches were presented which aid and support the development of agents for games. The approaches were identified by analysing potentially critical points in the development of agents through a literature survey and requirement interviews. The new POSH-SHARP architecture reduces the number of errors when including and modifying behaviours and their primitives. It also offers better support for tracking and iteratively developing behaviours and supports the division of labour into design and implementation. The extended ramp provides a "white box" solution for augmenting the selection process and requires minimal adjustments. Approaching agent design from an evolutionary perspective is not new. However, the case study using GP demonstrates for the first time the feasibility of using unmodified gameplay traces to drive the full evolution of agents from a clean slate to non-trivial behaviour. This demonstrates that genetic programming of game agents can be used as an option to explore novel directions for agent design. By combining all those elements and driven by the AGILE BEHAVIOUR DESIGN new systems can be developed or existing ones enriched in a more robust and guided way.

Appendix A

Behaviour-Oriented Design

Figure A-1 presented a reduced version of a pyPOSH Unreal Tournament agent behaviour. The behaviour is responsible for the agent movement and contains three different behaviour primitives. The two perceptual primitives know_enemy_base_pos and know_own_base_pos only return a boolean value in case the agent memorised the location in question. The action primitive to_enemy_flag has its own internal perception check and if the location of the enemy flag. If the location is known to the agent, it uses it tells the game to move the agent accordingly.

```
class Movement(Behaviour):
    def __init__(self, agent):
        Behaviour.__init__(self, agent,
                           ("to_enemy_flag"),
                           ("know_enemy_base_pos", "know_own_base_pos"))
        self.PosInfo = PositionsInfo()
        # set up useful constants
        self.PathHomeID = "PathHome"

    # === SENSES ===

    # returns 1 if we have a location for the enemy base
    def know_enemy_base_pos(self):
        #print "in know_enemy_base_pos sense"
        if self.PosInfo.EnemyBasePos == None:
            return 0
        else:
            return 1

    # returns 1 if we have a location for our own base
    def know_own_base_pos(self):
        if self.PosInfo.OwnBasePos == None:
            return 0
        else:
            return 1


    # === ACTIONS ===

    # runs to the enemy flag
    def to_enemy_flag(self):

        if self.PosInfo.has_enemy_flag_info_expired():
            self.PosInfo.expire_enemy_flag_info()

        if self.PosInfo.EnemyFlagInfo != {}:
            self.agent.Bot.send_message("RUNTO",
                {"Target" : self.PosInfo.EnemyFlagInfo["Id"]})
        return 1
```

Figure A-1: A reduced pyPOSH behaviour from the behaviour library for an Unreal Tournament agent. The behaviour is controlling an individual agent's movement within the environment and contains three behaviour primitives.

```
; This file was generated by A.B.O.D.E.
; Do not add comments to this file directly, as they may be
; lost the next time the tool is used.
; ---------------------------------------------------------------------
(
(documentation "StarCraft ThreeHatchHydra" "Swen E Gaudl" "initial plan")
(AP AP-inch (seconds 1.0) (   walk ))
(AP AP-moveing (minutes 1.0) ( moveto_navpoint ))
(C wander-map (seconds 20.0) (goal ((focusing_task 1.0 =)))
   (elements
      (  (CE-moveto-navpoint (trigger ((selected_target 1.0 =) (
          reached_target 0.0 =))) moveto-selected-nav)   )
      (  (CE-find-nxt-waypoint (trigger ((close_navpoint 1.0 =)))
          select_navpoint 1)   )
   )
)
(C get-enemy-flag (seconds 30.0) (goal ((have_enemy_flag 1.0 =)))
   (elements
      (  (CE-moveto-flag (trigger ((selected_target 1.0 =) (reached_target
           0.0 =))) moveto-selected-nav) )
      (  (CE-select-enemy-flag (trigger ((have_enemy_flag 1.0 !=)))
          select_enemy_flag) )
   )
)
(C retrace-back (minutes 1.0) (goal ((at_own_base 1.0 =)))
   (elements
      (  (CE-retrace-navpoint (trigger ((selected_target 1.0 =) (
          reached_target 0.0 =))) moveto-selected-nav)  )
      (  (CE-retrace-home (trigger ((close_navpoint 1.0 =)))
          retrace_navpoint)   )
   )
)
(C moveto-selected-nav (seconds 1.0) (goal ((reached_target 1.0 =)))
   (elements
      (  (CE-moveto-navpoint (trigger ((close_navpoint 1.0 =) (is_walking
          1.0 !=) (selected_navpoint_reachable 1.0 =))) moveto_navpoint)
          )
   )
)
(DC life (goal ((game_ended 1.0 =)))
   (drives
      ( (return-enemy-flag (trigger ((have_enemy_flag 1.0 =)))
          retrace-back(seconds 0.3)) )
      ( (get-enemy-flag-from-base (trigger ((enemy_flag_reachable 1.0 =)))
           get-enemy-flag(seconds 0.3)) )
      ( (wander_around (trigger ((focusing_task 1.0 !=))) wander-map(
          seconds 0.3)) )
      ( (inch (trigger ((succeed))) AP-inch(seconds 0.3)) )
   )
)
)
```

Figure A-2: A POSH plan for UNREAL TOURNAMENT agents on a capture the flag map. The plan is controlling a single agent.

Appendix B

StarCraft

In digital games a group of games, namely the strategy games, can benefit from a similar approach to AI design. The next section discusses the original design of AI for StarCraft using a dynamic planner whereas Section 4.6 discusses the notion of multiple strategies and how to transition between them based on the current state.

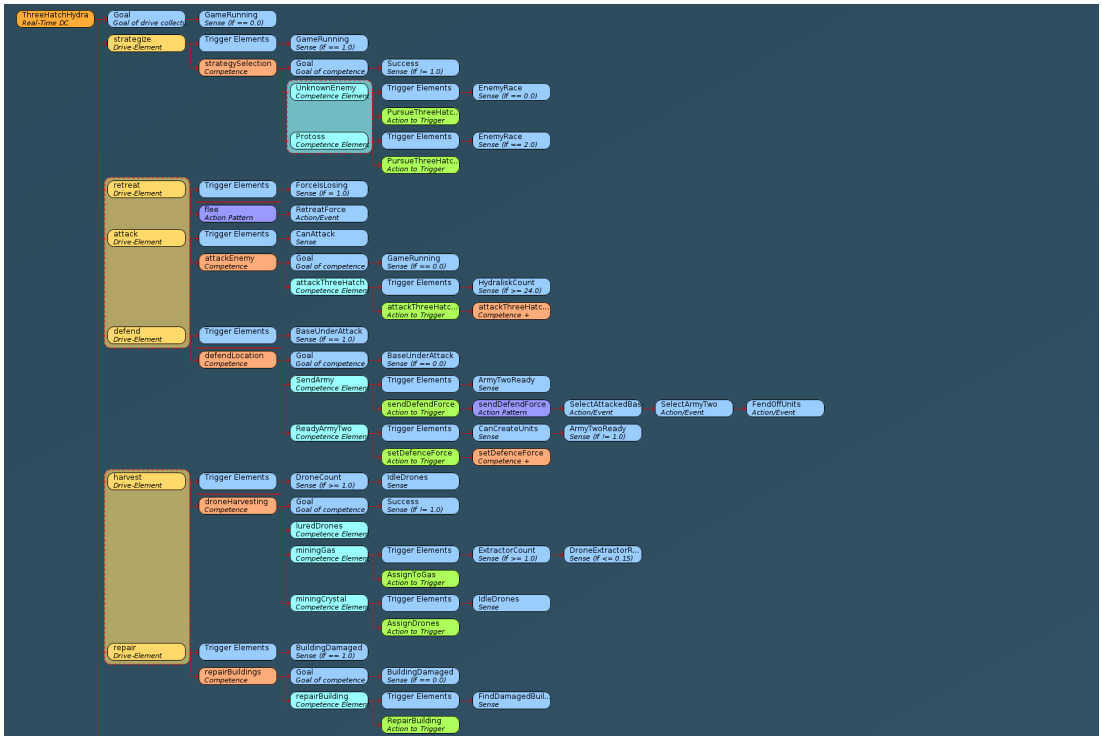


Figure B-1: The *ThreeHatchHydra* POSH plan for the extended StarCraft agent. The presented plan visualises the 11 Drives which form the agent and their contained competences. The drive elements are clustered based on their priority. Continued in figure B-2

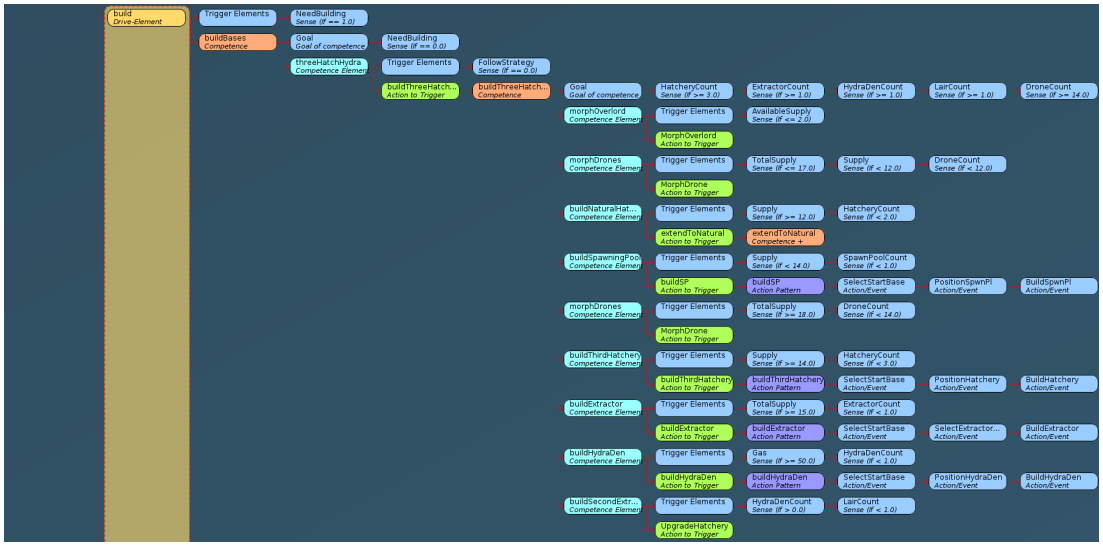


Figure B-2: The *ThreeHatchHydra* POSH plan is continued from figure B-1. However, drives of equal priority form groups which are surrounded by doted lines, e.g. harvest and repair. Continued in figure B-3

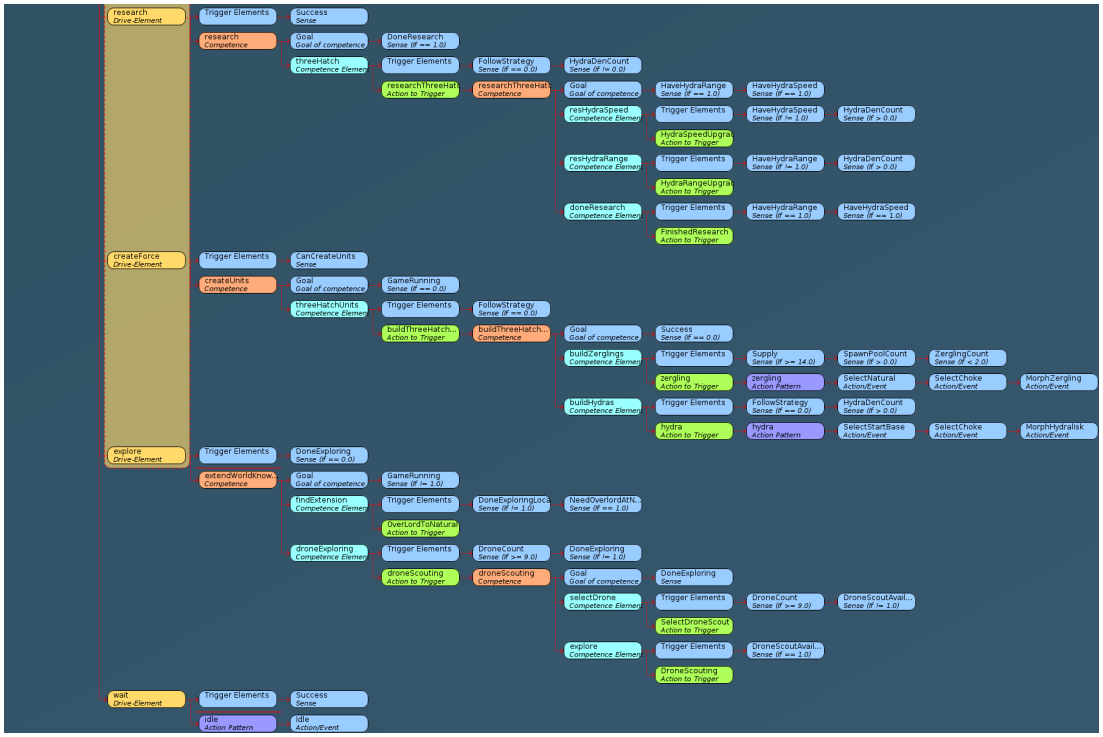


Figure B-3: The *ThreeHatchHydra* POSH plan is continued from figure B-2. The plan contains the full strategy to implement an original strategy from `http://wiki.teamliquid.net/starcraft/3_Hatch_Hydralisk_(vs._Protoss)`

Appendix C

Requirements

C.1 A Behaviour Language

In Chapter 3.6 the System-Specific Step was derived from analysing three IVA architectures and different points which are unique in the three inspected systems: Abl, FAtiMA and Bod were identified. From those common elements individual sss for each platform and overarching ones were deduced by conducting informal interviews with a set of 11 developers from different teams. In the following table those points are drawn together generating an overview over all sss.

The data represented in Tables C-1 & C-2 is derived by analysing information interview conducted with multiple teams and a major part of April Grow's research at the University of California, Santa Cruz. The interviews were either conducted in person or via skype sessions. During the first phase of the interviews the Lost Translator scenario was described and then implemented on an abstracted level by each team. After analysing the individual approaches, the teams were re-visited and the findings were presented and discussed to validate the findings.

#	Name	Summary	Systems	Authoring Support
i	Start Minimally	Having a working vertical slice early gives programmers and designers a good overview of the scenario structure	BOD/ POSH, ABL	Current ABODE* graphical design tool is sufficient
ii	Decompose Iteratively	Filling in the stubs iteratively gives designers and programmers freedom to adjust the structure without getting in each other's way	BOD/ POSH, ABL	Current ABODE* graphical design tool is sufficient
iii	Minimise and Encapsulate	The BOD/POSH tree relies on simple logic to execute quickly, so complex sensory preconditions should be offloaded to behaviours	BOD/ POSH	A module that manages encapsulated behaviours, keeping them simple and proposing them to new authors
iv	Goals First	The agent's actions are driven by goals, so there must always be a goal structure	FATIMA, BOD/ POSH	Combined with SSS Element v
v	Find Decision Points	Necessary scenario-defined decision points make sub-goals more apparent to author	FATIMA	Scenario event sequencing tool with prompts for goals and actions at decision points

Figure C-1: A summary of the SSS Elements described in the case study from chapter 3 and collected in a compressed form. The table is continued in figure C-2

#	Name	Summary	Systems	Authoring Support
vi	Goal Weighting and Tuning	Agent's different behaviours are driven by different weights, which is a huge time sink to debug	FAtiMA	Parallel execution and real-time adjustment/ comparison of values
vii	Intent Goals for Future Consequences	Language-specific limitations, such as only having one active goal at a time, hinter novice-intermediate authors	FAtiMA	Better documentation
viii	Define Coding Idioms	As ABL is its own language, an author must have a strong understanding of their chosen idioms before coding	ABL	Too advanced for a tool to offer much help
ix	NPC and Player Considerations	An author must conceptualise roles, the contents of the working memory and ABT, and fine-grain performance details while building up their behaviours	ABL	Revival of the ABL Debugger through modularisation: offline code analysis of behaviour structures through idioms
x	Consider Interruptions	Authors must try to make their behaviours robust against interruptions and stalling, which complicates the previous SSS Element	ABL, Bod	Revival of the ABL Debugger through modularisation: tree visualisation of iterations and disparate tree sections

Figure C-2: A summary of the SSS Elements described in the case study from chapter 3 and collected in a compressed form. The table a continuation from figure C-1.

Appendix D

Augmenting Action Selection Mechanisms

In Figure D-1 the first 6.6 seconds of a StarCraft using posh-sharp are represented. The agent is using the plan presented in Figure B-1 and is using the posh-sharp logging mechanism to record which elements of the plan are active and what they return. The log file also contains the initialisation of the agent and is recording the log in DEBUG mode which contains more information than the INFO or ERROR mode. The log also shows which behaviour are used and how they are connected to behaviour primitives.

```
1  [    343ms] DEBUG 0.DP.ThreeHatchHydra - Created
2  [    343ms] DEBUG 0.SDC.ThreeHatchHydra - Created
3  [    343ms] DEBUG 0          - Resetting the behaviours
4  [    343ms] DEBUG 0          - Resetting behaviour POSH_StarCraftBot.
       behaviours.UnitControl
5  [    343ms] DEBUG 0          - Resetting behaviour POSH_StarCraftBot.
       behaviours.StrategyControl
6  [    343ms] DEBUG 0          - Resetting behaviour POSH_StarCraftBot.
       behaviours.CombatControl
7  [    343ms] DEBUG 0          - Resetting behaviour POSH_StarCraftBot.Core
8  [   6599ms] DEBUG 0          - Resetting behaviour POSH_StarCraftBot.
       behaviours.BuildingControl
9  [   6599ms] DEBUG 0          - Resetting behaviour POSH_StarCraftBot.
       behaviours.ResourceControl
10 [   6599ms] DEBUG 0          - Waiting for behaviours ready
11 [   6599ms] DEBUG 0          - Behaviours ready
12 [   6599ms] DEBUG 0          - Reset successful
13 [   6599ms] DEBUG 0          - Starting real-time loop
14 [   6599ms] DEBUG 0          - Waiting for behaviours ready
```

Figure D-1: The first part of a log4Net log for the posh-sharp StarCraft agent. The log contains the initialisation and the first 6 seconds of a match. The log in continued in figure D-2.

```
[  6599ms] DEBUG 0           - Behaviours ready
[  6599ms] DEBUG 0           - Processing Drive Collection
[  6599ms] DEBUG 0.SDC.ThreeHatchHydra - Fired
[  6599ms] DEBUG 0.T.POSH_StarCraftBot.behaviours.StrategyControl.
    GameRunning - Firing
[  6599ms] DEBUG 0.Sense.GameRunning - Firing
[  6599ms] DEBUG 0.T.POSH_StarCraftBot.behaviours.StrategyControl.
    GameRunning - Sense POSH_StarCraftBot.behaviours.StrategyControl.
    GameRunning failed
[  6599ms] DEBUG 0.DP.ThreeHatchHydra - Fired
[  6615ms] DEBUG 0.T.POSH_StarCraftBot.behaviours.StrategyControl.
    GameRunning - Firing
[  6615ms] DEBUG 0.Sense.GameRunning - Firing
[  6615ms] DEBUG 0.DE.strategize - Fired
[  6615ms] DEBUG 0.C.strategySelection - Fired
[  6615ms] DEBUG 0.T.POSH_StarCraftBot.Core.Success - Firing
[  6615ms] DEBUG 0.Sense.Success - Firing
[  6615ms] DEBUG 0.T.POSH_StarCraftBot.Core.Success - Sense
    POSH_StarCraftBot.Core.Success failed
[  6615ms] DEBUG 0.CP.strategySelection - Fired
[  6615ms] DEBUG 0.T.POSH_StarCraftBot.behaviours.StrategyControl.
    EnemyRace - Firing
[  6615ms] DEBUG 0.Sense.EnemyRace - Firing
```

Figure D-2: The second part of a log4Net log for the POSH-SHARP STARCRAFT agent. The log contains the initialisation and the first 6 seconds of a match.

Figure D-3 demonstrates how the log file is visualised in INFO mode on an android device. presenting only the actions and which agent is triggering them. The log updates whenever an agent is firing a primitive and uses a scrolling representation—the information is added at the top and the last lines of the output is removed.

Figure D-3: The StealthierPOSH Android game illustrating the usage of the logging mechanism on the upper left side of the screenshot. The output contains 10 lines which update every seconds by adding new content ad the top and fading out old information at the bottom.

Glossary

Abl "A Behavior Language", or short ABL (pronounced "able") is a re-active planning language written in JAVA my Michael Mateas. It is most commonly known for being the AI system for *Façade* [Mateas and Stern, 2003].. 74, 76, 77, 91–94, 102, 103, 112, 117, 120, 121, 123–125, 127, 129–136, 141–145, 147, 160, 163, 173, 180, 215–218, 220, 230

Abode The advanced behaviour-oritended design editor, or short ABODE alows a visual development of posh plans. It is a stand-alone JAVA-based editor, accessible at `https://github.com/suegy/abode-star`.. 97, 98, 105, 106, 111, 123, 133, 145, 148, 154, 156, 157, 167

balance Balancing refers to the adjustment of difficulty, most of the times driven by feedback from testers of the target audience. If a game is well-balanced, players express the differences between forces or the difficulty of the game is appropriate.. 14

bwapi BWAPI offers an interface to access and change data within StarCraft. Thereby, it allows the inclusion of external code to represent artificial agents within the game. The API allows two modes which either offer the inclusion of an agent through a TCP/IP connection or through direct memory access when using a dynamic library file ("DLL"). More information are available at: `https://github.com/bwapi/bwapi`. 137, 141, 145, 146

deeper agent behaviour Deeper agent behaviour combines different aspects of what essentially is intended to describe more appealing or immersive agent behaviour for games. It is based on the definition of Deeper Gaming Experience by Burke et al. [2001]. Deeper agent behaviour combines:

- Situated: Agents are able to make decisions based on partial world knowledge.

- Reactive: Agents are able to respond in an appropriate amount of time to sudden changes in the Environment.
- Expressive: Agents have personalities and are able to represent them in interactions with environment and player.
- Sound: The player is able to attribute a "Theory of Mind" to the agent according to its actions.
- Scalable: The computational impact of the deeper agent behaviour has to be minimal to scale to many agents.

Each of the elements can be seen as dimensions which form a space for agents to be placed in. The original motivation for Deeper Gaming Experience is aimed at directing new research towards better designable agents, which this adjusted definition follows. . 1, 70, 95, 101, 102, 111, 114, 117, 154, 178–180, 209, 218

dms A computational component of an agent which simulates or emulates the process of a natural agents process making decisions. The process is also referred to as action selection with the contrast that action selection mostly refers to exibited behaviour and decion making is the internal process.. 69, 83, 117, 118, 121, 123, 137, 219, 220, 223

Eclipse The Eclipse foundation, `https://eclipse.org/`, provides a framework for java-based IDE creation. The most prominent one is the Eclipse Java-IDE which is widely used in industry and academia. Eclipse provides IDEs for most programming languages and the framework can use used to develop IDEs a new or special languages. The main focus on the Eclipse framework is modularity which allows easy recombination in tegration of new modules. The IDEs in the Eclipse framework are open-source and extentable and are available for all standard desktop operating systems. . 102–104

fps Frames per second, or short FPS, is a performance measure within game environments. The framerate gives a measure of how fluent the game runs on the used hardware. Typically, at a frame rate of 60FPS the user cannot see any disturbances of the game perframance. With older hardware and games the boundary was 30FPS to create a fluent impression of the visual representation.. 83, 146, 175

GameBots GameBots are a engine side interface developed by Adobbati et al. [2001]. The interface offers a serverside port to modify and communicate with objects

within the UNREAL TOURNAMENT game. Thereby, it offer a string-based communication protocol which communicates changes to and from the game in an asynchronous way.. 104

human-like Choi et al. [2007] defines human-like behaviour by following three main principles. An agent is human-like if it is using a similar processing structure as a human. The available sensory-motor system should be the same as a human would have access to. And human and agent share the same basic knowledge abouth the environment including the agents embodiment.. 85

immerse IMMERSE is a DARPA projected within the Strategic Social Interaction Modules program. The project goal is to develop a Unity-based training environment for soldiers to learn and reinforce "best-practice" skills when interacting with foreign cultures. The project is using autonomous characters and social stories which the users can experience to explore possible outcomes of his or her actions. The project is carried out by the Expressive Intelligence Group at the University of California. A more detailed description is available at: `https://users.soe.ucsc.edu/~mccoyjo/`.. 121, 122, 129, 131

IntelliJ IntelliJ is a propriatory IDE developed by jetbrains, `https://www.jetbrains.com`. The IDE comes in two flarours, ULTIMATE which comes with the latest features and dedictaed support and COMMUNITY which is freely available, does not integrate the latest fixes and only minimal support. The IDE is well designed and comes with good support for all major languages. Due to its closed development the usability is central to IntelliJ which makes it a versatile, robust and well maintained IDE. . 102

Jgap JGAP is a java-based evolutionary framework for developing and applying genetic algorithms and genetic programs to a variety of problems and more specifically it allows the inclusion of the evolutionary process into other java programs. It is developed and maintained by Meffert et al. [2000] and is open-source. JGAP provides a mechanism for evolving agents when given a set of command genes, a fitness function, a genetic selector and an interface to the target application.It is available at: `jgap.sourceforge.net/`. 202, 207, 221

log4Net Apache's Log4Net provides a standardised, configurable monitor support in the form of a modular logging architecture. Using XML based configuration files it is possible to set up monitor logs handling even large amounts of data.

Log4Net is a dynamic library for the Microsoft *.Net* architecture. It is available at: `https://logging.apache.org/log4net/`. 174

Microsoft Public License The Ms-PL is similar to other copyleft software licenses and is available at `https://opensource.org/licenses/MS-PL`. It allows the usage and distribution of the software but is less restrictive than the Gnu General Public License, see `http://www.gnu.org/licenses/gpl-3.0.en.html`.. 109

Netbeans Netbeans is another established IDE for java-based software development and similar to Eclipse features support for other languages as well. Netbeans, in contrast to Eclipse is owned by Oracle. Nonetheless, the project is open-source and it is it freely available for developers at `https://netbeans.org/`. Netbeans is maintained by Oracle and is less modular than Eclipse. It offers a stricter interface for plugins, which allows Netbeans to maintain or more directed user experience.. 102, 104

orts ORTS is a programming environment for developing real-time strategy AI and testing it. It is open-source and allows offline or online game sessions. More information are available at: `https://skatgame.net/mburo/orts`. 143

play trace Play traces are logged representations of interactions of a player within a given environment. Pay traces are beneficial for understand and analysing human player interaction within a game. The general understanding is that play traces go beyond simple player statistics but are more directed towards allowing develeopers either to replay what the player has done within the game environment or within some amalysis tool. . 22, 100, 214

posh POSH planning is a form of reactive planning [Ghallab et al., 2004], which offer faster planning times for agents. In contrast to traditional deliberative planning, the planner only plans the next action on the agents trajectory towards its global goal. POSH uses a parallel-rooted plan structure to introduce hierarchies and parallel execution into the plan which provides a compromise between responsive behaviour and goal directedness. . 7, 23, 24, 43, 95–99, 104–107, 111, 112, 123–126, 133, 135, 145–149, 151, 152, 154, 156–158, 160, 162, 165–172, 174–178, 189, 190, 203, 215, 220, 222, 223, 227–229, 231

posh-sharp The POSH-SHARP planner extends the original POSH system by integrating advanced modules for augmenting the selection and arbitration process. The architecture was developed for cross-platform agent development and works on

smartphones and in the browser. It is based on Microsoft's C# language and freely available in binary and source form at: `https://github.com/suegy/posh-sharp`. 1, 6, 21, 24, 115, 161, 168–170, 172–178, 200, 213, 215, 216, 218–221, 223, 224, 233, 234

Quake Quake is a commercial game series which released the first game in 1996. The games are developed by id Software and are fast paced. The game series focuses on a reaction driven play style and are considered one of the fastest first person shooters requiring the player to jump and evade other players while taking out other opponents. The games focus more on one-2-one encounters than on longer tactical play.. 86, 113

StarCraft StarCraft is a real-time strategy game developed and releaseb by Blizzard in 1998. The game became known for its well balanced game play and continues to attracted attention based on this. The game features three main races which have a unique skill set and require different strategies. Due to a freely available interface the game additionally became famous in the research community as it allows experimentation which advanced computational approaches in a challenging real-time environment. More information are available at: `http://us.blizzard.com/en-us/games/sc/`. 23, 47, 56, 57, 67, 76, 93, 133, 136, 137, 140–144, 146, 147, 152–157, 160, 161, 167, 175, 216, 220, 223, 228, 233, 234

Unreal Tournament Unreal Tournament (UT) is a commercial game series developed by EPIC. The games provides through a series of interfaces ways to integrate external libraries or inputs making them highly appealing to research due to their stable and expandable game infrastructure. The original games are first-person shooter requiring teams of players to combat each other. The games are more tactic driven than other games such as Quake.. 83, 98, 104, 105, 113, 223, 225–227, 238

wargus Wargus is real-time strategy game based on the mechanics of the game WarCraft by Blizzard. More details are available at: `http://wargus.sourceforge.net`. 141

Acronyms

Bod Behaviour-Oriented Design. 20, 21, 43, 94, 95, 97–100, 102, 105, 106, 112, 113, 116, 120, 122–125, 127, 129, 131–137, 141, 143, 144, 146, 148–150, 152–157, 159–165, 167, 178, 215–217, 220, 222, 223, 230–232

Bt BehaviorTree. 7, 37–46, 68, 70, 74–77, 92, 93, 95, 104, 107–111, 119, 133, 154, 163, 167, 168, 189, 203, 204, 215, 216, 219, 221, 223

CIG Computational Intelligence in Games. 137

dlc Downloadable Content. 10, 163

FAtiMA **F**earNot! **A**ffec**ti**ve **M**ind **A**rchitecture. 117, 124, 125, 127–129, 131, 132, 135, 136, 215, 216, 220, 230–232

fsm Finite-State Machine. 31–34, 65, 72, 73, 79, 111, 112, 114, 143, 149, 219, 223

gda Goal-Driven Autonomy. 112

Goap Goal-Oriented Planning. 77, 112, 158, 167

GWT Global Workspace Theory. 199

Ide Integrated Development Environment. 102, 103, 111, 135, 174, 219

IVA Interactive Virtual Agent. 1, 7, 21–23, 27, 69, 102, 115–119, 129, 133–136, 163, 174, 178, 180, 181, 200–204, 209, 212–215, 217–220, 222, 223, 230

mcts Monte-Carlo TreeSearch. 64–68, 142, 154, 222

rts Real-Time Strategy. 135, 136, 140, 141, 143, 154, 161, 162, 220, 223

sss System-Specific Step. 118–125, 127–130, 132–137, 161–163, 215, 222, 230–232

Bibliography

Adobbati, R., Marshall, A. N., Scholer, A., Tejada, S., Kaminka, G. A., Schaffer, S., and Sollitto, C. (2001). Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the second international workshop on Infrastructure for Agents, MAS, and Scalable MAS*, volume 5. Montreal, Canada.

Anderson, J. R. (1993). *Rules of the mind*. Psychology Press.

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., and Qin, Y. (2004). An integrated theory of the mind. *Psychological review*, 111(4):1036.

Anguelov, B. (2014). Synchronized behavior trees. `https://takinginitiative.wordpress.com/2014/02/17/synchronized-behavior-trees/`. last visited: 27. December 2015.

Argall, B. D., Chernova, S., Veloso, M., and Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469 – 483.

Arkin, R. C. (1998). *An Behavior-based Robotics*. MIT Press, Cambridge, MA, USA, 1st edition.

Assanie, M. (2002). Directable synthetic characters. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, pages 1–7.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multi-armed bandit problem. *Machine learning*, 47(2-3):235–256.

Baars, B. J. (2002). The conscious access hypothesis: origins and recent evidence. *Trends in cognitive sciences*, 6(1):47–52.

Bäck, T. (1996). *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford Univ. Press, New York, NY.

Bäck, T., Fogel, D. B., and Michalewicz, Z. (2000). *Evolutionary computation 1: Basic algorithms and operators*, volume 1. CRC Press.

Barraquand, J. and Latombe, J.-C. (1991). Robot motion planning: A distributed representation approach. *The International Journal of Robotics Research*, 10(6):628–649.

Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10):70–77.

Becroft, D., Bassett, J., Mejía, A., Rich, C., and Sidner, C. L. (2011). Aipaint: A sketch-based behavior tree authoring tool. In *AIIDE*.

Beetz, M., Mösenlechner, L., Tenorth, M., and Rühr, T. (2012). Cram – a cognitive robot abstract machine. In *5th International Conference on Cognitive Systems (CogSys 2012)*.

Bengio, Y. (2009). Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127.

Bernardini, S. and Porayska-Pomsta, K. (2013). Planning-based social partners for children with autism. In *Twenty-Third International Conference on Automated Planning and Scheduling*.

Bethke, E. (2003). *Game Development and Production*. Wordware game developer's library. Wordware Pub.

Boden, M. A. (1998). Creativity and artificial intelligence. *Artificial Intelligence*, 103(1):347–356.

Bojic, I., Lipic, T., Kusek, M., and Jezic, G. (2011). Extending the jade agent behaviour model with jbehaviourtrees framework. In *Agent and Multi-Agent Systems: Technologies and Applications*, pages 159–168. Springer.

Bourg, D. and Seemann, G. (2004). *AI for Game Developers*. O' Reilly. O'Reilly, first edition edition.

Branavan, S. R. K., Silver, D., and Barzilay, R. (2011). Non-linear monte-carlo search in civilization ii. In Walsh, T., editor, *IJCAI*, pages 2404–2410. IJCAI/AAAI.

Brandy, L. (2010). lbrandy.com blog archive using genetic algorithms to find starcraft 2 build orders. website: `http://lbrandy.com/blog/2010/11/using-genetic-algorithms-to-find-starcraft-2-build-orders`. [Accessed 3rd Dec 2011].

Broekens, J., Heerink, M., and Rosendal, H. (2009). Assistive social robots in elderly care: a review. *Gerontechnology*, 8(2):94–103.

Brom, C., Gemrot, J., Bída, M., Burkert, O., Partington, S. J., and Bryson, J. J. (2006). POSH tools for game agent development by students and non-programmers. In Mehdi, Q., Mtenzi, F., Duggan, B., and McAtamney, H., editors, *The Ninth International Computer Games Conference: AI, Mobile, Educational and Serious Games*, pages 126–133. University of Wolverhampton.

Brooks, R. (1986). A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23.

Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, 47(13):139–159.

Brown, J. W. and Nee, D. E. (2012). Executive control of cognitive search. In Todd et al. [2012], chapter Search, Goals, and the Brain, pages 69–80.

Browne, C., Powley, E. J., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43.

Bryson, J. J. (2000a). Cross-paradigm analysis of autonomous agent architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 12(2):165–190.

Bryson, J. J. (2000b). Hierarchy and sequence vs. full parallelism in reactive action selection architectures. In *From Animals to Animats 6 (SAB00)*, pages 147–156, Cambridge, MA. MIT Press.

Bryson, J. J. (2001). *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD thesis, MIT, Department of EECS, Cambridge, MA. AI Technical Report 2001-003.

Bryson, J. J. (2003). The Behavior-Oriented Design of modular agent intelligence. In Kowalszyk, R., Müller, J. P., Tianfield, H., and Unland, R., editors, *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*, pages 61–76. Springer, Berlin.

Bryson, J. J. (2005). Modular representations of cognitive phenomena in AI, psychology and neuroscience. In Davis, D. N., editor, *Visions of Mind: Architectures for Cognition and Affect*, pages 66–89. Idea Group.

Bryson, J. J. (2012). A role for consciousness in action selection. *International Journal of Machine Consciousness*, 04(02):471–482.

Bryson, J. J. and McGonigle, B. (1998). Agent architecture as object oriented design. In Singh, M. P., Rao, A. S., and Wooldridge, M. J., editors, *The Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL97)*, pages 15–30, Providence, RI. Springer.

Bryson, J. J. and Stein, L. A. (2001). Modularity and design in reactive intelligence. In *Proceedings of the* 17^{th} *International Joint Conference on Artificial Intelligence*, pages 1115–1120, Seattle. Morgan Kaufmann.

Bryson, J. J. and Thórisson, K. R. (2000). Dragons, bats & evil knights: A three-layer design approach to character based creative play. *Virtual Reality*, 5(2):57–71.

Burke, R., Isla, D., Downie, M., Ivanov, Y., and Blumberg, B. (2001). Creature smarts: The art and architecture of a virtual brain. In *Proceedings Game Developers Conference*, pages 1–20.

Buro, M. (2003). Real-time strategy games: A new ai research challenge. In *IJCAI*, pages 1534–1535.

Buro, M. and Churchill, D. (2012). Real-time strategy game competitions. *AI Magazine*, 33(3):106.

BWAPI Development Team (2010). bwapi — an API for interacting with starcraft: Broodwar (1.16.1) - google project hosting. website: `http://code.google.com/p/bwapi/`. [Accessed 4th Nov 2011].

Chaimowicz, L. and Kumar, V. (2007). Aerial shepherds: Coordination among uavs and swarms of robots. In *Distributed Autonomous Robotic Systems 6*, pages 243–252. Springer.

Champandard, A. (2007a). Behavior trees for Next-Gen game AI. website: `http://aigamedev.com/open/article/behavior-trees-part1/`. [Accessed 4th Apr 2012].

Champandard, A. J. (2003). *AI Game Development*. New Riders Publishing.

Champandard, A. J. (2007b). Assaulting f.e.a.r.s ai: 29 tricks to arm your game. `http://aigamedev.com/open/review/fear-ai/`. last visited: 3. November 2015.

Champandard, A. J. (2007c). Living with the sims ai: 21 tricks to adopt for your game. `http://aigamedev.com/open/highlights/the-sims-ai/`. last visited: 3. November 2015.

Champandard, A. J. (2007d). Understanding behavior trees. `http://aigamedev.com/open/article/bt-overview/`. last visited: 3. December 2015.

Champandard, A. J. (2008). Getting started with decision making and control systems. In Rabin, S., editor, *Ai Game Programming Wisdom 4*, volume 4 of *Cengage Learning*, chapter 3 Architecture, pages 257–264. Charles River Media, Inc.

Champandard, A. J. (2012). Making designers obsolete? evolution in game design. `http://aigamedev.com/open/review/evolution-in-cityconquest/`. last visited: 3. November 2015.

Champandard, A. J. and Dunstan, P. (2013). The behavior tree starter kit. In Rabin, S., editor, *Game AI Pro: Collected Wisdom of Game AI Professionals*, Game Ai Pro, pages 72–92. A. K. Peters, Ltd.

Chandler, H. (2009). *The Game Production Handbook*. Computer science series. Infinity Science Press.

Choi, D., Kaufman, M., Langley, P., Nejati, N., and Shapiro, D. (2004). An architecture for persistent reactive behavior. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 988–995. IEEE Computer Society.

Choi, D., Könik, T., Nejati, N., Park, C., and Langley, P. (2007). A believable agent for first-person shooter games. In *AIIDE*, pages 71–73.

Clark, C. and Storkey, A. (2014). Teaching deep convolutional neural networks to play go. *arXiv preprint arXiv:1412.3409*.

Colton, S., Wiggins, G. A., et al. (2012). Computational creativity: the final frontier? In *ECAI*, volume 12, pages 21–26.

Cook, M., Colton, S., and Gow, J. (2014). Automating game design in three dimensions. In *Proceedings of the AISB Symposium on AI and Games*, pages 20–24.

Cools, R. (2012). Chemical neuromodulation of goal-directed behavior. In Todd et al. [2012], chapter Search, Goals, and the Brain, pages 111–125.

Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In Ciancarini, P. and van den Herik, H. J., editors, *Proceedings*, number 5 in International Conference on Computer and Games, pages 72–83. Springer.

Couzin, I. D., Ioannou, C. C., Demirel, G., Gross, T., Torney, C. J., Hartnett, A., Conradt, L., Levin, S. A., and Leonard, N. E. (2011). Uninformed individuals promote democratic consensus in animal groups. *science*, 334(6062):1578–1580.

Davies, S. (2012). Development of an ai for the real-time-strategy game starcraft using behaviour oriented design. Bachelor's thesis, Department of Computer Science, University of Bath.

Dias, J., Mascarenhas, S., and Paiva, A. (2014). Fatima modular: Towards an agent architecture with a generic appraisal framework. In *Emotion Modeling*, pages 44–56. Springer.

Dias, J. and Paiva, A. (2011). Agents with emotional intelligence for storytelling. In *Affective Computing and Intelligent Interaction*, pages 77–86. Springer.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

dkollmann (2011). Brainiac designer. `https://brainiac.codeplex.com`. last visited: 25. December 2015.

D'Mello, S. K., Franklin, S., Ramamurthy, U., and Baars, B. J. (2006). A cognitive science based machine learning architecture. In *AAAI Spring Symposium: Between a Rock and a Hard Place: Cognitive Science Principles Meet AI-Hard Problems*, pages 40–45. AAAI.

Downie, M. (2005). *Choreographing the extended agent*. PhD thesis, MIT Media Lab, Massachusetts Institute of Technology.

Dromey, R. G. (2003). From requirements to design: Formalizing the key steps. In *Software Engineering and Formal Methods, 2003. Proceedings. First International Conference on*, pages 2–11. IEEE.

Dybsand, E. (2003). Ai middleware: Getting into character, part 1: Ai implant. `http://www.gamasutra.com/view/feature/131234/ai_middleware_getting_into_.php`. last visited: 25. December 2015.

Evans, R. (forthcoming). Computer models of social practices. In *Synthese Library: Philosophy and Theory of Artificial Intelligence*. Synthese.

Feng, D. and Krogh, B. H. (1986). Acceleration-constrained time-optimal control in n dimensions. *Automatic Control, IEEE Transactions on*, 31(10):955–958.

Franklin, S., Baars, B. J., and Ramamurthy, U. (2009). Robots need conscious perception: A reply to aleksander and haikonen. In *NEWSLETTER ON PHILOSOPHY AND COMPUTERS*, volume 09, pages 13–15. APA.

Franklin, S. and Patterson Jr, F. (2006). The lida architecture: Adding new modes of learning to an intelligent, autonomous, software agent. *pat*, 703:764–1004.

Gaudl, S. E. and Bryson, J. J. (2014). Extended ramp goal module: Low-cost behaviour arbitration for real-time controllers based on biological models of dopamine cells. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE.

Gaudl, S. E., Davies, S., and Bryson, J. J. (2013). Behaviour oriented design for real-time-strategy games – an approach on iterative development for starcraft ai. In *Proceedings of the Foundations of Digital Games*, pages 198–205. Society for the Advancement of Science of Digital Games.

Gaudl, S. E., Osborn, J. C., and Bryson, J. J. (2015). Learning from play: Facilitating character design through genetic programming and human mimicry. In *Progress in Artificial Intelligence*, pages 292–297. Springer International Publishing.

Ge, S. S. and Cui, Y. J. (2000). New potential functions for mobile robot path planning. *IEEE Transactions on robotics and automation*, 16(5):615–620.

Gemrot, J., Kadlec, R., Bída, M., Burkert, O., Píbil, R., Havlíček, J., Zemčák, L., Šimlovič, J., Vansa, R., Štolba, M., Plch, T., and C., B. (2009). Pogamut 3 can assist developers in building ai (not only) for their videogame agents. In *Agents for Games and Simulations*, number 5920 in LNCS, pages 1–15. Springer.

Georgeff, M. P. and Lansky, A. L. (1987). Reactive reasoning and planning. In *AAAI*, volume 87, pages 677–682.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated planning: theory & practice*. Elsevier.

Gomes, P. and Jhala, A. (2013). Ai authoring for virtual characters in conflict. In *Proceedings on the Ninth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Grand, S., Cliff, D., and Malhotra, A. (1997). Creatures: Artificial life autonomous software agents for home entertainment. In Johnson, W. L., editor, *Proceedings of the First International Conference on Autonomous Agents*, pages 22–29. ACM press.

Grey, J. and Bryson, J. J. (2011). Procedural quests: A focus for agent interaction in role playing games. In Romano, D. and Moffat, D., editors, *Proceedings of the AISB 2011 Symposium: AI & Games*, pages 3–10, York. SSAISB.

Grow, A. (2015). Enabl: A modular authoring interface for creating interactive characters. Advancement report, University of California, Santa Cruz.

Grow, A., Gaudl, S. E., Gomes, P. F., Mateas, M., and Wardrip-Fruin, N. (2014). A methodology for requirements analysis of ai architecture authoring tools. In *Foundations of Digital Games 2014*. Society for the Advancement of the Science of Digital Games.

Hagelbäck, J. (2012). Potential-field based navigation in StarCaft. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 388–393. IEEE.

Hagelbäck, J. and Johansson, S. J. (2008). Using multi-agent potential fields in real-time strategy games. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 631–638. International Foundation for Autonomous Agents and Multiagent Systems.

Hamed (2012). Skill. `https://skill.codeplex.com/`. last visited: 25. December 2015.

Harbour, J. S. (2004). *Game Programming All in One*. Gale virtual reference library. Thomson Course Technology.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107.

Hecker, C. (2009). My liner notes for spore/spore behavior tree docs. `http://chrishecker.com/My_liner_notes_for_spore/Spore_Behavior_Tree_Docs`. last visited: 14. December 2015.

Holmgard, C., Liapis, A., Togelius, J., and Yannakakis, G. (2014). Evolving personas for player decision modeling. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558.

Huang, H. (2011). Skynet meets the swarm: how the berkeley overmind won the 2010 StarCraft AI competition. `http://arstechnica.com/gaming/news/2011/`

01/skynet-meets-the-swarm-how-the-berkeley-overmind-won-the-2010-starcraft-ai-competition.ars. [Accessed 11th Nov 2011].

Huizinga, J. (1950). *Homo ludens-a study of the play element in culture*. Beacon press.

Isla, D. (2005). GDC 2005 proceeding: Handling complexity in the halo 2 AI. http://www.gamasutra.com/view/feature/2250/gdc_2005_proceeding_handling. [Accessed 4th Apr 2012].

Isla, D., Burke, R., Downie, M., and Blumberg, B. (2001). A layered brain architecture for synthetic creatures. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 1051–1058. IJCAI.

Johansen, E. (2013). Behave. http://angryant.com/behave/. last visited: 25. December 2015.

Justesen, N., Tillman, B., Togelius, J., and Risi, S. (2014). Script-and cluster-based uct for starcraft. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE.

Kadlec, R., Gemrot, J., Burkert, O., Bída, M., Havlíček, J., and Brom, C. (2007). Pogamut 2–a platform for fast development of virtual agents behaviour. In *Proceedings of CGAMES'2007*, volume 7. Citeseer.

Kasparov, G. (2016). Game changers. *New Scientist*, 229(3063):26–27.

Keith, C. (2010). *Agile Game Development with Scrum*. Addison-Wesley Signature Series (Cohn). Pearson Education.

Kelley, T. and Kelley, D. (2013). *Creative confidence: Unleashing the creative potential within us all*. Crown Business.

Khatib, F., Cooper, S., Tyka, M. D., Xu, K., Makedon, I., and Baker, D. (2011). Algorithm discovery by protein folding game players. In vol. 108 no. 47, editor, *Proceedings of the National Academy of Sciences of the United States of America*, volume 108, pages 18949–18953.

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. *The international journal of robotics research*, 5(1):90–98.

Kocsis, L., Szepesvári, C., and Willemson, J. (2006). Improved monte-carlo search. Technical report, University of Tartu.

Koenig, S., Likhachev, M., Liu, Y., and Furcy, D. (2004). Incremental heuristic search in ai. *AI Magazine*, 25(2):99–112.

Konolige, K. (2000). A gradient method for realtime robot control. In *Intelligent Robots and Systems, 2000.(IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, volume 1, pages 639–646. IEEE.

Konolige, K. and Myers, K. (1998). The Saphira architecture for autonomous mobile robots. In Kortenkamp, D., Bonasso, R. P., and Murphy, R., editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, chapter 9, pages 211–242. MIT Press, Cambridge, MA.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press.

Krawiec, K. and O'Reilly, U.-M. (2014). Behavioral programming: a broader and more detailed take on semantic gp. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 935–942. ACM.

Krogh, B. and Thorpe, C. (1986). Integrated path planning and dynamic steering control for autonomous vehicles. In *Robotics and Automation. Proceedings. 1986 IEEE International Conference on*, volume 3, pages 1664–1669.

Kulawardana, D. (2011). DiLIB behavior tree library. `http://dilib.dimutu.com/`. last visited: 15. December 2015.

Laird, J. E. (2001). Using a computer game to develop advanced ai. *Computer*, 34(7):70–75.

Laird, J. E., Assanie, M., Bachelor, B., Benninghoff, N., Enam, S., Jones, B., Kerfoot, A., Lauver, C., Magerko, B., Sheiman, J., et al. (2000). A test bed for developing intelligent synthetic characters. *Ann Arbor*, 1001:48109–2110.

Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artif. Intell.*, 33(1):1–64.

Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986). Chunking in soar: The anatomy of a general learning mechanism. *Machine learning*, 1(1):11–46.

Laird, J. E. and van Lent, M. (2000). Human-level ai's killer application: Interactive computer games. In Kautz, H. A. and Porter, B. W., editors, *AAAI/IAAI*, pages 1171–1178. AAAI Press / The MIT Press.

LaMothe, A. (2000). A neural-net primer. *Game Programming Gems*, 1:330–350.

Langley, P., Laird, J. E., and Rogers, S. (2009). Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160.

Langley, P., McKusick, K. B., Allen, J. A., Iba, W. F., and Thompson, K. (1991). A design for the icarus architecture. *ACM SIGART Bulletin*, 2(4):104–109.

Laugier, C. and Fraichard, T. (2001). Robot control architecture and motion autonomy. In Vlacic, L., Parent, M., and Harashima, F., editors, *Intelligent Vehicle Technologies: Theory and Applications*, Automotive Engineering Series, chapter Decisional Architectures for Motion Autonomy, pages 334–348. Butterworth-Heinemann.

Lavalle, S. M. (1998). Rapidly-exploring random trees: A new tool for path planning. Technical report, Iowa State University.

Lim, C.-U., Baumgarten, R., and Colton, S. (2010). *Applications of Evolutionary Computation: EvoApplicatons 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC, Istanbul, Turkey, April 7-9, 2010, Proceedings, Part I*, chapter Evolving Behaviour Trees for the Commercial Game DEFCON, pages 100–110. Springer Berlin Heidelberg, Berlin, Heidelberg.

Loyall, B., Bryan, A., and Bates, L. J. (1991). Hap a reactive, adaptive architecture for agents. Tech-Report CMU-CS-91-147, Carnegie Mellon University.

Lucas, S. M. (2005). Evolving a neural network location evaluator to play ms. pac-man. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 1–8. IEEE.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. (2005). Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527.

Madison, D. (2005). *Process mapping, process improvement, and process management: a practical guide for enhancing work and information flow.* Paton Professional.

Maes, P. (1993). Behavior-based artificial intelligence. In *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, pages 74–83.

Magerko, B., Laird, J., Assanie, M., Kerfoot, A., and Stokes, D. (2004). Ai characters and directors for interactive computer games. *Ann Arbor*, 1001(48):109–2110.

Mali, A. D. (2002). On the behavior-based architectures of autonomous agency. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 32(3):231–242.

Mark, D. (2009). *Behavioral Mathematics for Game AI*. Applied Mathematics Series. Course Technology Cengage Learning.

Mark, D. (2010). Improving ai decision modeling through utility theory. `http://gdcvault.com/play/1012410/Improving-AI-Decision-Modeling-Through`. last visited: 14. September 2015.

Mark, D. (2012). Ai architectures: A culinary guide. *Game Developer Magazine*, 19(8):7–12.

Martin, C. B. and Deuze, M. (2009). The independent production of culture: A digital games case study. *Games and Culture*, 4(3):276–295.

Mateas, M. (2002). *Interactive Drama, Art, and Artificial Intelligence*. Technical report cmu-cs-02-206, School of Computer Science, Carnegie Mellon University.

Mateas, M. (2003). Expressive ai: Games and artificial intelligence. In *DIGRA Conf.*

Mateas, M. and Stern, A. (2002). A behavior language for story-based believable agents. *Intelligent Systems, IEEE*, 17(4):39–47.

Mateas, M. and Stern, A. (2003). Façade: An experiment in building a fully-realized interactive drama. In *Game Developers Conference*, volume 2.

Mathes, W. F., Brownley, K. A., Mo, X., and Bulik, C. M. (2009). The biology of binge eating. *Appetite*, 52(3):545–553.

Mathes, W. F., Nehrenberg, D. L., Gordon, R., Hua, K., Garland, T., and Pomp, D. (2010). Dopaminergic dysregulation in mice selectively bred for excessive exercise or obesity. *Behavioural brain research*, 210(2):155–163.

McCoy, J. and Mateas, M. (2008). An integrated agent for playing real-time strategy games. In *AAAI*, volume 8, pages 1313–1318.

McCoy, J., Treanor, M., Samuel, B., Reed, A. A., Mateas, M., and Wardrip-Fruin, N. (2013). Prom week: Designing past the game/story dilemma. In *FDG*, pages 94–101.

Meffert, K., Rotstan, N., Knowles, C., and Sangiorgi, U. (2000). Jgap-java genetic algorithms and genetic programming package. `http://jgap.sf.net`. last viewed:01.2015.

Millington, I. and Funge, J. (2009a). *Artificial Intelligence for Games*. Morgan Kaufmann, second edition edition.

Millington, I. and Funge, J. (2009b). *Tactical Analysis*, chapter 6 Tactical and Strategic AI, pages 518–553. In Millington and Funge [2009a], second edition edition.

Molineaux, M., Klenk, M., and Aha, D. W. (2010). Goal-driven autonomy in a navy strategy simulation. Technical report, DTIC Document.

Montfort, N. and Bogost, I. (2009). Pac-man. In *Racing the Beam:The Atari Video Computer System*, pages 65–79. MIT Press.

Müller, M. (2002). Computer go. *Artificial Intelligence*, 134(12):145 – 179.

Muñoz-Avila, H., Aha, D. W., Jaidee, U., Klenk, M., and Molineaux, M. (2010). Applying goal driven autonomy to a team shooter game. In Guesgen, H. W. and Murray, R. C., editors, *Proceedings of the Twenty-Third International Florida Artificial Intelligence Research Society Conference*. AAAI Press.

Murray, J. (2004). From game-story to cyberdrama. *First person: New media as story, performance, and game*, 1:2–11.

Nelson, M. J. and Mateas, M. (2009). A requirements analysis for videogame design support tools. In *Proceedings of the Fourth International Conference on the Foundations of Digital Games*, pages 137–144.

Newell, A. (1994). *Unified theories of cognition*. Harvard University Press.

Novak, J. (2011). *Game Development Essentials: An Introduction*. Cengage Learning.

O'Donnell, C. (2009). The everyday lives of video game developers: Experimentally understanding underlying systems/structures. *Transformative Works and Cultures*, 2.

O'Donnell, C. (2012). This is not a software industry. In Zackariasson, P. and Wilson, T., editors, *The Video Game Industry: Formation, Present State, and Future*, chapter 1, pages 17–33. Routledge, New York, NY, 10001.

Olsen, J. M. (2002). Attractors and repulsors. In *Game Programming Gems 4*. Charles River Media, Inc., Rockland, MA, USA.

O'Neil, M. and Ryan, C. (2003). Grammatical evolution. In *Grammatical Evolution*, pages 33–47. Springer.

Orkin, J. (2004). Symbolic representation of game world state: Toward real-time planning in games. In Fu, D., Henke, S., and Orkin, J., editors, *Proceedings of*

the AAAI Workshop on Challenges in Game Artificial Intelligence, volume 5, pages 26–30, Menlo Park, CA. AAAI Press.

Orkin, J. (2005). Agent architecture considerations for real-time planning in games. In Young, M. R. and John, L., editors, *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 105–110, Menlo Park, CA. AAAI Press.

Ortega, J., Shaker, N., Togelius, J., and Yannakakis, G. N. (2013). Imitating human playing styles in super mario bros. *Entertainment Computing*, 4(2):93 – 104.

Osborn, J. C. and Mateas, M. (2014). A game-independent play trace dissimilarity metric. In *Proceedings of the Foundations of Digital Games*. Society for the Advancement of Science of Digital Games.

Paiva, A. (2013). Gaips: Intelligent agents and synthetic characters.

Partington, S. J. (2005). A critical analysis of behaviour-oriented design (BOD), based on experiences in using it to create an unreal tournament capture-the-flag (CTF) team. Undergraduate Dissertation, University of Bath.

Partington, S. J. and Bryson, J. J. (2005). The Behavior Oriented Design of an Unreal Tournament character. In Panayiotopoulos, T., Gratch, J., Aylett, R., Ballin, D., Olivier, P., and Rist, T., editors, *The Fifth International Working Conference on Intelligent Virtual Agents*, pages 466–477, Kos, Greece. Springer.

Perez, D., Nicolau, M., ONeill, M., and Brabazon, A. (2011). Evolving behaviour trees for the mario ai competition using grammatical evolution. In Di Chio, e., editor, *Applications of Evolutionary Computation*, volume 6624 of *Lecture Notes in Computer Science*, pages 123–132. Springer Berlin Heidelberg.

Poli, R., Langdon, W. B., McPhee, N. F., and Koza, J. R. (2008). *A field guide to genetic programming*. Lulu. com.

Poli, R. and McPhee, N. F. (2008). Covariant parsimony pressure in genetic programming. Technical report, Citeseer.

Rabin, S. (2010a). *Artificial Intelligrence: Agents, Architecture, and Techniques*, chapter Game Programming: Graphics, Animation, AI, Audio, and Networking, pages 521–558. In Rabin [2010b].

Rabin, S. (2010b). *Introduction to Game Development: Second Edition*. Game development series. Course Technology Cengage Learning.

Rao, A. S. and Georgeff, M. P. (1991). Modeling rational agents within a bdi-architecture. *KR*, 91:473–484.

Raskin, J. (1994). Viewpoint: Intuitive equals familiar. *Commun. ACM*, 37(9):17–18.

Rauwolf, P., Mitchell, D., and Bryson, J. J. (2015). Value homophily benefits cooperation but motivates employing incorrect social information. *Journal of theoretical biology*, 367:246–261.

Redish, A. D. (2012). Search processes and hippocampus. In Todd et al. [2012], chapter Search, Goals, and the Brain, pages 81–96.

Rohlfshagen, P. and Bryson, J. J. (2010). Flexible latching: A biologically-inspired mechanism for improving the management of homeostatic goals. *Cognitive Computation*, 2(3):230–241.

Rosenbloom, P. S., Laird, J., and Newell, A., editors (1993). *The SOAR papers: Research on integrated intelligence*. Mit Press Cambridge, MA.

Rubenstein, M., Cornejo, A., and Nagpal, R. (2014). Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799.

Rubin, K. (2012). *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Signature Series (Cohn). Pearson Education.

Russell, S. J. and Norvig, P. (1995). *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Sandberg, T. and Togelius, J. (2011). Evolutionary Multi-Agent potential field based AI approach for SSC scenarios in RTS games. Master's thesis, IT University Copenhagen.

Schmidt, C. (2015). Dev blog #27: Ai in battle brothers, part 1. `http://battlebrothersgame.com/dev-blog-27-ai-battle-brothers-part-1/`. last visited: 25. December 2015.

Schwab, B. (2004). *Ai Game Engine Programming (Game Development Series)*. Charles River Media, Inc., Rockland, MA, USA.

Schwefel, H.-P. (1993). *Evolution and optimum seeking: the sixth generation*. John Wiley & Sons, Inc.

Shanahan, M. (2006). A cognitive architecture that combines internal simulation with a global workspace. *Consciousness and cognition*, 15(2):433–449.

Shantia, A., Begue, E., and Wiering, M. (2011). Connectionist reinforcement learning for intelligent unit micro management in starcraft. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1794–1801. IEEE.

Shapiro, D., McCoy, J., Grow, A., Samuel, B., Stern, A., Swanson, R., Treanor, M., and Mateas, M. (2013). Creating playable social experiences through whole-body interaction with virtual characters. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Simon, H. A. (1972). Theories of bounded rationality. *Decision and organization*, 1(1):161–176.

Smit, S. K. and Eiben, A. E. (2009). Comparing parameter tuning methods for evolutionary algorithms. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 399–406. IEEE.

Snavely, P. (2004). Empowering design: Defining fuzzy logic behaviour through excel-based spreadsheets. In Rabin, S., editor, *AI programming Wisdom 2*, volume 2, chapter 9.4 Scripting, pages 541–548. Charles River Media, Inc., Hingham, MA.

Snook, G. (2000). Simplified 3d movement and pathfinding using navigation meshes. *Game Programming Gems*, 1:288–304.

Soemers, D. (2014). Tactical planning using mcts in the game of starcraft1. Technical report, Maastricht University.

Spierling, U. and Szilas, N. (2009). Authoring issues beyond tools. In *Interactive Storytelling*, pages 50–61. Springer.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2005). Real-time neuroevolution in the nero video game. *Evolutionary Computation, IEEE Transactions on*, 9(6):653–668.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127.

Stentz, A. (1994). Optimal and efficient path planning for partially-known environments. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 3310–3317. IEEE.

Stephens, D. W. and Krebs, J. R. (1986). *Foraging theory*. Princeton University Press.

Stewart, T. C., Bekolay, T., and Eliasmith, C. (2012). Learning to select actions with spiking neurons in the basal ganglia. *Frontiers in Neuroscience*, 6(2):1–14.

Stirling, L. (2010). *Game Writing and Interactive Storytelling*, chapter Game Design, pages 139–164. In Rabin [2010b].

Storey, M.-A., Fracchia, F. D., and Müller, H. A. (1999). Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185.

Sweetser, P. M. (2004a). How to build neural networks for games. In Rabin, S., editor, *AI Programming Wisdoms*, volume 2, pages 615–625. Charles River Media, Inc., Hingham, MA.

Sweetser, P. M. (2004b). Strategic decision-making with neural networks and influence maps. In Rabin, S., editor, *AI Programming Wisdoms*, volume 2, pages 439–446. Charles River Media, Inc., Hingham, MA.

Taylor, T. L. (2012). *Raising the Stakes: E-Sports and the Professionalization of Computer Gaming*. The MIT Press.

Thompson, T. (2014). Arkham intelligence. `http://aiandgames.com/arkham-intelligence/`. last visited: 14. December 2015.

Thorn, A. (2013). *Game Development Principles*. Game Development. Cengage Learning.

Todd, P., Hills, T., and Robbins, T. (2012). *Cognitive Search: Evolution, Algorithms, and the Brain*. Strüngmann Forum reports. University Press Group Limited.

Togelius, J., Karakovskiy, S., and Baumgarten, R. (2010). The 2009 mario ai competition. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE.

Togelius, J. and Lucas, S. M. (2005). Evolving controllers for simulated car racing. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, pages 1906–1913. IEEE.

Togelius, J., Yannakakis, G., Karakovskiy, S., and Shaker, N. (2012). Assessing believability. In Hingston, P., editor, *Believable Bots*, pages 215–230. Springer Berlin Heidelberg.

Tomlinson, W. M. (2002). *Synthetic social relationships for computational entities*. PhD thesis, Massachusetts Institute of Technology.

Tozour, P. (2001). Influence mapping. *Game Programming Gems*, 2:287–297.

Tremblay, J., Torres, P. A., Rikovitch, N., and Verbrugge, C. (2013). An exploration tool for predicting stealthy behaviour. In *Proceedings of the 2013 AIIDE Workshop on Artificial Intelligence in the Game Design Process, IDP*, volume 2013.

Tyrrell, T. (1993). *Computational mechanisms for action selection*. PhD thesis, University of Edinburgh Edinburgh, Scotland.

University, C. M. (2013a). CTAT: Cognitive tutor authoring tools.

University, R. (2013b). CADIA: Center for analysis and design of intelligent agents.

van der Werf, E. C., Van Den Herik, H. J., and Uiterwijk, J. W. (2003). Solving go on small boards. *ICGA Journal*, 26(2):92–107.

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 National Conference on Artificial Intelligence*, page 929931.

Van Velsen, M. (2008). Narratoria, an authoring suite for digital interactive narrative. In *FLAIRS Conference*, pages 394–395.

Velsquez, J. D. (1998). Modeling emotion-based decision-making.

Veres, S. M., Lincoln, N. K., and Molnar, L. (2011). Control engineering of autonomous cognitive vehicles-a practical tutorial.

Wallis, A. (2007). Tooling around: Pathfinding with kynogon's kynapse. `http : / / www.gamasutra.com / view / news / 104269 / Tooling_Around_Pathfinding_With_Kynogons_Kynapse.php`. last visited: 25. November 2015.

Ward, C. D. and Cowling, P. I. (2009). Monte carlo search applied to card selection in magic: The gathering. In Lanzi, P. L., editor, *CIG*, pages 9–16. IEEE.

Weber, B., Mateas, M., and Jhala, A. (2010a). Applying Goal-Driven autonomy to StarCraft. In *Proceedings of the Sixth Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Weber, B., Mawhorter, P., Mateas, M., and Jhala, A. (2010b). Reactive planning idioms for multi-scale game ai. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 115–122.

Weber, B. G., Mateas, M., and Jhala, A. (2011). Building human-level ai for real-time strategy games. In *AAAI Fall Symposium: Advances in Cognitive Systems*, volume 11, page 01.

Weber, B. G., Mawhorter, P., Mateas, M., and Jhala, A. (2010c). Reactive planning idioms for multi-scale game AI. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 115–122. IEEE.

Wen, L. and Dromey, R. G. (2004). From requirements change to design change: A formal path. In *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 104–113. IEEE.

Wiggins, G. A. (2006). Searching for computational creativity. *New Generation Computing*, 24(3):209–222.

Wintermute, S., Xu, J., and Laird, J. E. (2007). Sorts: A human-level approach to real-time strategy ai. *Ann Arbor*, 1001(48):109–2121.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. Wiley, second edition edition.

Wooldridge, M., Jennings, N. R., et al. (1995). Intelligent agents: Theory and practice. *Knowledge engineering review*, 10(2):115–152.

Yannakakis, G. N. (2012). Game ai revisited. In *Proceedings of the 9th conference on Computing Frontiers*, pages 285–292. ACM.

Zackariasson, P. (2013). The role of creativity. In Hotho, S. and McGregor, N., editors, *Changing the Rules of the Game: Economic, Management and Emerging Issues in the Computer Games Industry*, EBL-Schweitzer, chapter 6, pages 105–121. Palgrave Macmillan.

Zimmermann, H.-J. (2001). *Fuzzy Set Theoryand Its Applications*. Springer Netherlands.