POSH-SHARP: A Lightweight ToolKit for Creating Cognitive Agents

Swen E. Gaudl[1]

Abstract. Agent design is an intricate process requiring skills from different disciplines. Thus experts in one domain are not necessarily experts in the others. Supporting the design of agents is important and needs to address varying skill and expertise as well as varying handling the design of complex agents. In this paper, a new agent design toolkit –POSH-SHARP– for intelligent virtual agents (IVAs) and cognitive embodied agents is presented. It was designed to address the need for a robust agent development framework in highly restrictive environments such as the web or smartphones while being useful to both novice and expert users. It includes advanced functionality such as debug support, explicit design rules using a related design methodology and a simple set-up and distribution mechanism to reduce the authoring burden for large iteratively developed agents. The new framework was implemented in C# and contains sample code for different game environments to offer novice users a starting point.

1 Introduction

This paper presents a new agent modelling toolkit and framework for designing intelligent virtual agents (IVAs) in games which offers affordances such as an easy set-up and distribution within industrial game environments, support for debugging and a low computational overhead.

Non player characters (NPCs) in games can range from simple entities that respond with a pre-determined reply such as giving the player a quest from a stack of quests to embodied cognitive agents that respond based on the players behaviour and the state of the world. In the first instance —the simple agent— finite state machines (FSMs) can be sufficient for modelling the behaviour of such entities. The agent does only select from a stack of quests an item and returns it to the player and might vary the reply sentence based on a list of pre-written replies. However, when the game world and the response patterns have to be more complex, more sophisticated approaches might be required. State machines present a very visual, easy way of modelling behaviour, which makes them appealing to designers in contrast to decision tables or rule-based approaches. The downside f a state machine is that the number of transitions and changes to the underlying model do not scale well for large systems. Hierarchical state machines (HFSMs) aid in this situation marginally as they offer levels of abstraction and detail to model the behaviour of an entity on different granularities. Based on HFSMs, BehaviorTree (BT) [5, 7] became a dominant approach in the games industry as the approach scales well, can be visualised well and has a low computational overhead. In academia, more experimental approaches were developed such as ways to model more expressive agents using ABL[16] or to model cognitive processes more closely in FATIMA [8]. Due to the increasing capabilities of personal computers, existing cognitive frameworks were also used to model agent behaviour such Soar [20]. A similar approach to BT for modelling behaviour was developed by Bryson [3] with a focus on agent-based modelling in Science. Bryson integrated a design methodology with a LISP-like language *–posh–* and planner to allow novice programmers to model complex agent processes in a more guided way.

A novel framework and planner –POSH-SHARP– for designingPOSH agents is presented which extends the capabilities of its predecessor JYPOSH and offers new mechanisms of building and maintaining complex cognitive agents for virtual environments.

The rest of this paper is organised as follows. In Section 2, the context of the new system and how it positions itself within similar approaches is given. The system and its core components are described in Section 3 which included examples from existing agent implementations. The paper is finalised by a discussion of future work and open challenges.

2 Background & Related Work

Digital Games are more than software systems, they are cultural artefacts and artworks as well and are often highly interactive. Thus, designing and building games requires support beyond software engineering. SCRUM for Games [13] was developed to aid the design of games from a technical perspective but stays at a high abstraction level not supporting the design of its components, e.g. the AI system controlling characters. Agile Behaviour Design (A-Bed) [9] discusses an approach for aiding the design process of character AI and supplies a process model for developing agents, addressing this need for more fine-grained support. The presented toolkit uses but is not limited to A-Bed as a design method.

For games driven by a story or relying on the interaction between player and agent, agent design is a crucial part requiring a deep understanding of the game mechanics as well as the intended plot of the game. If done badly, agents can destroy the entire experience of a game by being either boring, too repetitive, obviously cheating, or un-responsive. One mechanism to develop less rigid agents is the use of planning systems such as GOAP [17]. Planners require expert authors to design the initial restrictions for a given domain. The planner then at runtime uses domain knowledge to predict ad plan possible behaviours to achieve the designed goal. This reduces the interdependence of nodes and the amount of manual checking transition for an author as they do not need to check all possible combinations when designing agent goals. This allows the resulting agent to scale well when designing separate goals incrementally. Mateas proposed an approach for writing complex, branching interactive drama for

[1] MetaMakers Institute, UK, email: swen.gaudl@gmail.com

games using a planning system — ABL[15]. In Façade this system monitors the responses and actions of the player and directs the story based on *story beats* in a certain way to create a novel and interesting experience. [19] uses the same approach to control a set of managers to create an agent for real-time strategy games. The advance of using a planning approach is that the system can respond to unforeseen changes and is very customisable and scales well even for highly complex agents. The downside of ABL is that the setup is complex and it requires a high level of skill to develop and maintain agents as the author needs to be both an expert in the domain of the game as well as an expert in planning systems. Due to the runtime creation of the agent and its changing representation, agents designed with able are hard to debug or inspect. An alternative to designed and planned behaviours is the use of cognitive approaches to model agents and then use those to drive to the story. This approach produces more *IMPROV-style* games or art installations such as AlphaWolf [12], an installation which simulates the behaviour of a wolf pack offering player interaction. Cognitive agent approaches offer an entirely new opportunity for scalable agent design as the designer only models individual agents in terms of their motivations and how they perceive and interact with the environment and other agents. Thus, the agents have to reason individually of how to achieve their goal reducing partially the complexity of pre-specifying each interaction. However, cognitive systems such as Isla et al.'s c4 system, or Sorts[20], a cognitive real-time strategy player, require a lot of computation resources as well as a thorough understanding of cognitive modelling. Because of these two reasons more sophisticated systems never transitioned into actual practice.

After introducing BT as a way of designing and structuring agent behaviour beyond state machines, Isla worked on a more applicable system working – the *F.E.A.R.* system [6]. Their system integrates a reactive planning system with lazy evaluation of memory[2] to allow for more performance but still heavily relies on experts when designing plans but offers better support in terms of tool support and computational resources.

3 POSH-SHARP

POSH is a lightweight reactive planning language offering a similar way to structuring behaviours to BT. However, it uses a separation between plan and agent implementation to decouple the platform-independent design of the plan with the platform-dependent implementation of the agent's actions, senses and memory within a given system or game.

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 reused 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 Behaviour Design (A-Bed), 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. Similar to BT design tools such as SKILL STUDIO[3], DI-LIB[4] and BRAINIAC DESIGNER[5],POSH used the ABODE editor to support designers when writing plan flies.

[2] In F.E.A.R. sensory information and memory is only updated every few frames to amortise the computational costs. When not updated the previous information is presented instead requiring no computation.

[3] `https://skill.codeplex.com/`

[4] `http://dilib.dimutu.com/`

[5] `http://brainiac.codeplex.com/`

To enhance the support of game AI development, a new arbitration architecture is proposed – POSH-SHARP– 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 integrated better 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 as all required libraries are dynamically linked. 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[6]. 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[7] system required a complex setup for individual machines and relied on access to system variables of the operating system. It also required the developer to maintain a complex folder structure which contained all sources and compiled code for both 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* files and in Python *egg* files. This helps reduce the burden of a programmer to maintain a manageable code base.

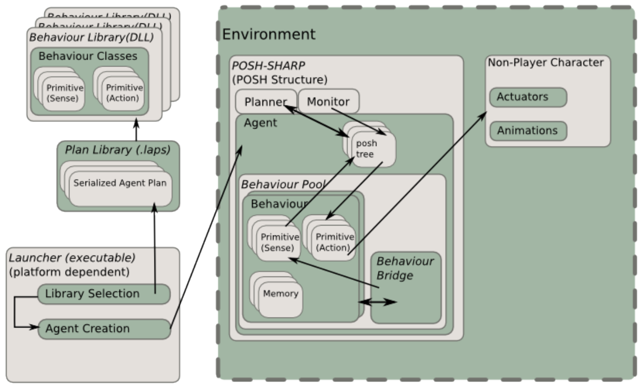


Figure 1: The POSH-SHARP architecture once the modules have been integrated into an environment, e.g. the integration with a game engine such as Unity.

[6] 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`

[7] `http://www.cs.bath.ac.uk/~jjb/web/pyposh.html`

3.1 POSH-SHARP Modules

Figure1 illustrates a view of the new layout of POSH-SHARP modules within a system and includes a view of how it integrates into an environment such as a game engine.

- 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[8] a specific launcher exists and integrates fully into the game engine.
- The **core** module is platform independent and can be used "as-is" as it does not rely on other modules, see POSH_SHARP(POSH Structure) in Figure 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, 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 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.

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

```
[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 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`

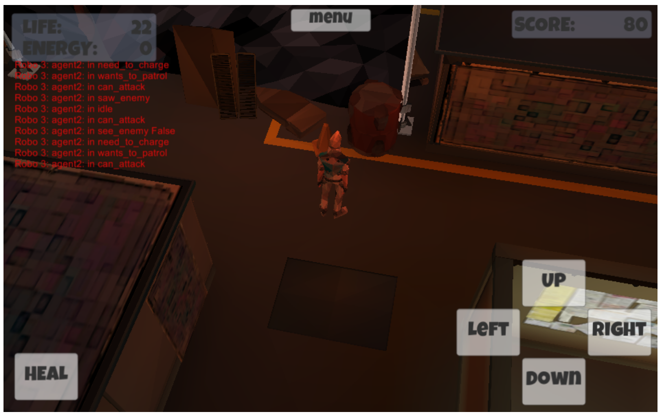
necessary, it reduces the visibility of own behaviours and increases the chance of modifying or removing essential parts of POSH unwillingly. POSH-SHARP introduces the packaged POSH *core*, combining the planner and the entire structure of the system into a 111kB sized dynamic library file. Behaviour files are also compiled into behaviour library DLLs. This is supported by free tools such as Xamarin's Monodevelop[9]. Upon starting POSH-SHARP, the core receives as a parameter a list of dynamic libraries which should be inspected.

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.

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

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

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



```
Robo 3: agent2: in need_to_charge
Robo 3: agent2: in wants_to_patrol
Robo 3: agent2: in can_attack
Robo 3: agent2: in saw_enemy
Robo 3: agent2: in idle
Robo 3: agent2: in can_attack
Robo 3: agent2: in see_enemy False
Robo 3: agent2: in need_to_charge
Robo 3: agent2: in wants_to_patrol
Robo 3: agent2: in can_attack
```

Figure 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.

, both augment a method and attach a name reference allowing the planner to search for them by the name and a version number. In Figure 2 an example action from the STEALTHIER POSH Android game, see Figure 3,is given which is using POSH-SHARP. The primitive is called by the planner when the robot agent needs to recharge the battery and uses a NavMesh[18] to identify if the agent is spatially close to a charger. To follow AB-ED, primitives should be as independent as possible and use their perception to reduce interdependencies. In this case, checking the internal state of the NavMesh. By offering the planner to inspect and search for possible primitives instead of providing them as a list when coding a behaviour library, 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 runtime 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, overloading existing primitives at design-time is a powerful process which allows developers to extend functionality by following the idea of Brook's SUBSUMPTION idiom in a real-time manner. It also offers more customisation option to a designer as behaviours can be swapped in and out.

3.3 Memory & Encapsulation

Similar to architectures such as ACT-R[1] and Soar[14], 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 1 provides 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 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 [17] or the *Working Memory Elements* of ABL, introduces extra complexity into the design process which 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 1. The bridge stores similar to the cX system[12], 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 number 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 [4] 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.

3.4 Monitoring Execution

As identified by Grow et al.[11] in their analysis of three intelligent agent frameworks the need for logging and debugging functionality is integrated into POSH-SHARP; the analysis 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[10]. 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 the system to switch between two modes of logging. The full log support using LOG4[11] or no 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 of possible behaviours which want to execute in size [2] 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 a marker in the internal tree representation is kept and the execution shifts it further down the tree when a drive is called. Instead of pushing a stack this mechanism reduces the allocation costs of spawning unneeded pointers.

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. To be able to analyse multiple runs of a long execution, POSH-SHARP writes a continuous-rolling log 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 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[12].

4 Future Work

The current POSH-SHARP toolkit has been tested in multiple scenarios ranging from StarCraft agents [10] to mobile games such as the previously mentioned STEALTHIER POSH. However, further feedback from professional developers in combination with experiments in industrial settings are still required to examine potential weaknesses of the system. The dynamic primitive switching of primitives which was introduced into POSH-SHARP is a complex process and needs further investigation and feedback from designers and testers to make it as useful as possible without affecting the creative freedom of an author. Visual representations of what agents do and how their reasoning process can be represented are crucial to the development of complex behaviour. The current visualisation and other forms of using the log provide potential directions for future research. The current approach to editing and visualising plan files using ABODE is an already identified shortcoming of the toolkit because the editor does not offer support beyond plan creation and visualisation. Additionally, a new approach for modelling and presenting parallel drive collections and their impact on each other is required, if the planner wants to compete with more sophisticated cognitive approaches. The current memory model provided by the *Behaviour Bridge* is a first step towards more cognitive and scalable models for agents. Nonetheless, this model is not able to compete with complex memory models in ACT-R and SOAR when using learning mechanisms to alter and evolve posh plans. A new version of memory that can be inspected by a designer might be a possible direction for future work as well.

5 Conclusion

To aid the development and to focus on multi-platform development the new POSH-SHARP arbitration architecture was proposed which is based on Bryson's original concept of POSH and extends it by four new features: multi-platform integration, behaviour inspection, behaviour versioning and the *Behaviour Bridge*. The idea behind POSH-SHARP is similar to the original concept of POSH still and additionally aims to provide a light-weight, flexible and modular approach to designing cognitive agents but increases the usability of the software by reducing potential problem points. POSH-SHARP introduces the behaviour library DLL, the core library and the launcher, which reduces the number of files to three and creates an easier to maintain a project. It simplifies the design process by automatically inspecting library files and extracting all behaviours and behaviour primitives requested by an agent. This reduces the impact of typos or wrongly

[10] This issue leads game developers to be cautious when using new approaches or approaches which allow for learning.

[11] 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. It is available at `https://logging.apache.org/log4net/`

[12] An illustration of the visual logging mechanism in STEALTHIERPOSH is available in Figure 3, page 4.

associated/non-existing primitives in behaviours. POSH-SHARP introduces a modular logging and debugging mechanism which allows a developer to trace the flow of information through the POSH graph aiding the developer while debugging and helping them create a robust agent system. The internal mechanisms such as the *Behaviour Bridge* and the *behaviour versioning* increase the capabilities of POSH and remove inter-dependencies between behaviours, The new mechanisms support robust incremental changes to behaviours. Future research directions for the toolkit have been identified and offer potential to expand the capabilities of the framework in different directions.

The combination of POSH-SHARP and AGILE BEHAVIOUR DESIGN is intended to support novice developers by guiding their design and giving them a robust and helpful set of development tools. The approach also allows expert developers to profit from explicit design steps and advanced support which can be used to verify the progress of a current project.

REFERENCES

[1] John Robert Anderson, *Rules of the mind*, Psychology Press, 1993.

[2] Joanna J. Bryson, *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*, Ph.D. dissertation, MIT, Department of EECS, Cambridge, MA, June 2001. AI Technical Report 2001-003.

[3] Joanna J. Bryson, 'The Behavior-Oriented Design of modular agent intelligence', in *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*, eds., R. Kowalszyk, Jörg P. Müller, H. Tianfield, and R. Unland, 61–76, Springer, Berlin, (2003).

[4] Robert Burke, Damian Isla, Marc Downie, Yuri Ivanov, and Bruce Blumberg, 'Creature smarts: The art and architecture of a virtual brain', in *Proceedings Game Developers Conference*, pp. 1–20, (2001).

[5] Alex J. Champandard, *AI Game Development*, New Riders Publishing, 2003.

[6] Alex J. Champandard. Assaulting f.e.a.r.s ai: 29 tricks to arm your game. `http://aigamedev.com/open/review/fear-ai/`, 10 2007. last visited: 3. November 2015.

[7] Alex J. Champandard and Philip Dunstan, 'The behavior tree starter kit', in *Game AI Pro: Collected Wisdom of Game AI Professionals*, ed., Steve Rabin, Game Ai Pro, 72–92, A. K. Peters, Ltd., (2013).

[8] Joao Dias, Samuel Mascarenhas, and Ana Paiva, 'Fatima modular: Towards an agent architecture with a generic appraisal framework', in *Emotion Modeling*, 44–56, Springer, (2014).

[9] Swen Gaudl, 'Agile behaviour design: A design approach for structuring game characters and interactions', in *Internation Conference on Digital Storytelling: Authoring Tool Workshop*, (2017).

[10] Swen E. Gaudl, Simon Davies, and Joanna J. Bryson, 'Behaviour oriented design for real-time-strategy games – an approach on iterative development for starcraft ai', in *Proceedings of the Foundations of Digital Games*, pp. 198–205. Society for the Advancement of Science of Digital Games, (2013).

[11] April Grow, Swen E. Gaudl, Paulo F. Gomes, Michael Mateas, and Noah Wardrip-Fruin, '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, (2014).

[12] Damian Isla, Robert Burke, Marc Downie, and Bruce Blumberg, 'A layered brain architecture for synthetic creatures', in *International Joint Conference on Artificial Intelligence*, volume 17, pp. 1051–1058. IJCAI, (2001).

[13] Clinton Keith, *Agile Game Development with Scrum*, Addison-Wesley Signature Series (Cohn), Pearson Education, 2010.

[14] John E. Laird, Allen Newell, and Paul S. Rosenbloom, 'Soar: An architecture for general intelligence', *Artif. Intell.*, **33**(1), 1–64, (1987).

[15] Michael Mateas, *Interactive Drama, Art, and Artificial Intelligence*, Technical report cmu-cs-02-206, School of Computer Science, Carnegie Mellon University, December 2002.

[16] Michael Mateas and Andrew Stern, 'A behavior language for story-based believable agents', *Intelligent Systems, IEEE*, **17**(4), 39–47, (2002).

[17] Jeff Orkin, 'Agent architecture considerations for real-time planning in games.', in *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference*, eds., Michael R. Young and Laird John, pp. 105–110, Menlo Park, CA, (2005). AAAI Press.

[18] Greg Snook, 'Simplified 3d movement and pathfinding using navigation meshes', *Game Programming Gems*, **1**, 288–304, (2000).

[19] B.G. Weber, P. Mawhorter, M. Mateas, and A. Jhala, 'Reactive planning idioms for multi-scale game ai', in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pp. 115–122, (2010).

[20] Sam Wintermute, Joseph Xu, and John E Laird, 'Sorts: A human-level approach to real-time strategy ai', *Ann Arbor*, **1001**(48), 109–2121, (2007).