A Methodology for Requirements Analysis of AI Architecture Authoring Tools

April Grow[1], Swen Gaudl[2], Paulo Gomes[1], Michael Mateas[1], Noah Wardrip-Fruin[1]

[1] Center for Games and Playable Media
University of California, Santa Cruz
Santa Cruz, CA, 95064
{agrow, pfontain, michaelm, nwf}
@soe.ucsc.edu

[2] Department of Computer Science
University of Bath
BA2 7AY, Bath, UK
s.e.gaudl @bath.ac.uk

ABSTRACT

Authoring embodied, highly interactive virtual agents (IVAs) for robust experiences is an extremely difficult task. Current architectures for creating those agents are so complex that it takes enormous amounts of effort to craft even short experiences, with lengthier, polished experiences (e.g., Facade, Ada and Grace) often requiring person-years of effort by expert authors. However, each architecture is challenging in vastly different ways; it is impossible to propose a universal authoring solution without being too general to provide significant leverage. Instead, we present our analysis of the *System-Specific Step* (SSS) in the IVA authoring process, encapsulated in the case studies of three different architectures tackling a simple scenario. The case studies revealed distinctly different behaviors by each team in their SSS, resulting in the need for different authoring solutions. We iteratively proposed and discussed each team's SSS Components and potential authoring support strategies to identify actionable software improvements. Our expectation is that other teams can perform similar analyses of their own systems' SSS and make authoring improvements where they are most needed. Further, our case-study approach provides a methodology for detailed comparison of the authoring affordances of different IVA architectures, providing a lens for understanding the similarities, differences and tradeoffs between architectures.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence – *intelligent agents, multiagent systems*; D.2.1 [**Software Engineering**]: Requirements/Specifications – *elicitation methods, methodologies*.

General Terms

Design, Standardization.

Keywords

Design, Agent Authoring, Interactive Virtual Agents, Tool-Driven Development, Behavior-Oriented Design, ABL

1. INTRODUCTION

Interactive Virtual Agents (IVAs) are embodied human characters that richly respond to user interaction, combining work in AI, interfaces, sensing technology, and graphics, as well as interdisciplinary knowledge from fields as diverse as psychology and theater. We have begun to see the uses of IVAs manifest across many fields, including medicine [3], human care-giving [12], education [14], interactive drama [15], and video games [10, 16]. There seem to be as many approaches to creating virtual embodied agents as there are humans creating them, but the majority of approaches have one thing in common: *authoring*.

Our particular use of the overloaded term *authoring* encompasses any asset creation and modification necessary to produce the desired functionality of IVAs: animation, audio, written dialogue, as well as behaviors, goals, and other more specialized decision-making constructs belonging to a custom decision-making mechanism (DMM). Authoring for the DMM adds another dimension of complexity to the authorial burden of IVAs beyond scripted scenes. The combinatoric interaction possibilities, including large internal (to the agent) and external state spaces, makes it difficult for an author to reason about and modify a DMM.

Authoring tools are often proposed as a means to help the author manage this complexity. However, authoring tools, especially for IVAs exhibiting complex behavior, are a research area of their own. To be of any practical use, authoring tools must be domain-specific, system-specific, or customizable enough to be tailored to the authoring challenges of a specific DMM; this fact is illustrated by the lack of any cross-architecture tools for IVAs. The choice to invest time and energy in an authoring tool is a difficult one, because it is challenging to determine if the cost of creating/customizing a tool and training authors would be less than brute-forcing the authoring challenge without the tool.

To better understand IVA authoring, we surveyed 11 IVA authors across 5 institutions and 9 different projects. We then returned to three teams for iterative interviews, where we discovered a similar pattern of difficulty, which we have decided to call the *System-Specific Step* (SSS). In the SSS lies the DMM-dependent tangled web of architectural affordance and constraint in which the author painfully translates their high-level vision for the character into a decision policy expressed in a specific architecture. We returned to the three teams with our interpretation of their system's SSS, confirming the requirements it places on any authoring tool approach, together with authoring tool proposals based on the challenges discovered in the SSS to gauge their reactions.

This paper proposes the SSS requirements analysis methodology as a means by which programmer-authors may better understand their specific system's authoring burden and potential features of authoring tools which would alleviate this burden. Further, our approach to requirements analysis provides a methodology for comparing multiple IVA architectures to better understand the relative strengths and tradeoffs of different architectures, as well as the different authoring metaphors and behavioral idioms supported by different architectures. We report three case studies of IVA architectures having different design philosophies, teams, and levels of complexity as rigorous example test cases of our methodology. Our goal is to inform the creation of authoring tools in similar architectures to enable the authoring of more robust IVAs, to potentially identify common patterns of authoring difficulty across architectures, and to provide a methodology for more rigorous comparison of the strengths, weaknesses and tradeoffs between different IVA architectures.

1.1 Related Work

The authors' are not aware of any other work that documents and analyzes the processes of different IVA authoring approaches over a common scenario. In [11], FatiMA and ABL, two target architectures of our case studies described below, were compared regarding their expressiveness for modeling conflict between characters. While related, this work focused on the output of the two architectures, rather than their authoring processes. Even though the content matter of the tools was different, Nelson and Mateas' iterative case studies regarding video game design support tools provided a compelling example for our IVA architecture authoring analysis [18]. We were able to build up a methodology and test it with our subjects through tight collaboration, which we feel was key to our success.

We have implicitly narrowed our definition of authors to programmer-authors in this paper; the designers with an authorial vision who have enough technical knowledge to build or use complicated or technical authoring tools. We would like to support less tech-savvy authors in the future, for which the list of authoring issues identified by Spierling and Szilas [21] also provides a useful starting point. The process of defining the SSS and tools supporting it involved iterative discussions with our intended authors in order to "make tools that better match the concepts and practices of [our] media designers and content creators" [21].

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 [24]. 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 was minimal, the addition of the Narratoria tool suite to the agent authoring process reduced the time spent authoring between two similar projects by half. 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 specialized authors, informed our conceptualization of the SSS.

AIPaint, a behavior tree authoring tool, gains its authoring power by limiting the behavioral domain to spatial reasoning [1]. In our case studies, we focused on the authoring of social behavior, as such behavior is characteristic of many IVA applications. Different metaphors and conventions will need to be supported for social behavior than for the spatial behavior supported by AIPaint. Finally, as AI research progresses, commercial AI systems in games also evolve using techniques from research to empower their systems. For example, as the needed complexity of game agents exceeded that which can be readily authored by finite state machine approaches, behavior trees [7] arose as one of the dominant commercial approaches to structuring and controlling intelligent agents in games. Two of the architectures examined in our case studies make use of reactive planning, which is closely related to behavior trees.

2. THE SYSTEM-SPECIFIC STEP

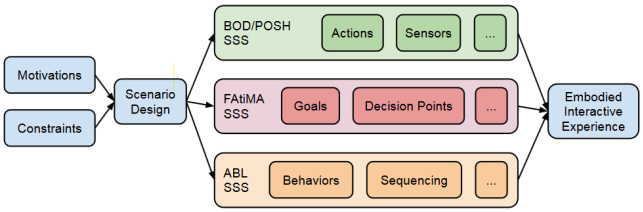

Figure 1. Illustrates the boundaries of an architecture's SSS

We acknowledge as a first-class authoring challenge that each IVA architecture comes with its own design philosophy, coding style, and data structures. While we can all share the idea of an authoring burden, how this burden manifests in each system can be entirely unique. In order to begin alleviating the authoring burden for each system, we need to identify the peculiarities of the authoring burden in specific instances.

The System-Specific Step is our term for the parts of the authoring process where broad discussion ends and, as the name suggests, system-specific design constructs are used instead. Any aspect of authoring that is driven by the commitments of a specific architecture, including design constructs and design philosophy, is part of the SSS of that system. Examples of what an author may need to do as part of the SSS include: imagining how an agent will traverse the behavior representation so as to iteratively author interesting decision points, constructing hierarchical goals so that an agent can plan its way from the beginning to end of the scenario, and figuring out how an agent can express frustration if its body is busy doing other actions. *Figure 1* shows a graphical representation of the boundaries of the SSSs using this paper' three case studies as examples, including a few of each system's unique primitives and tasks.

2.1 The SSS Conception

We conducted a series of informal interviews with 5 institutions across the globe to help understand how different institutions making use of different architectures approached their personal IVA authoring challenges. In addition to the six local ABL authors and the authors of this paper, we surveyed members of GAIPS [19], CADIA [23], and CTAT [22] to explore different approaches and purposes for authoring, anecdotes of successes and failures of particular authoring tools, and techniques for visualization. Our findings helped us propose the idea of the System-Specific Step to capture the architecture-specific phases of the authoring process, and led us to the methodology of using in-depth case studies to inform the design of authoring tools.

2.2 The SSS Methodology

In order to find a specific system's point of divergence from shared authoring concepts, we needed to run the architecture through an authoring exercise. For this paper, we designed a simple scenario, described below, for intermediate-expert authors to transform into descriptive pseudocode for their own system, one step removed from actually programming the scenario. These intermediate-expert authors were accompanied by an analyst who was not an expert in the architecture, which forced the intermediate-expert to elaborate and make explicit every step of their authoring process.

The authoring team then translated their work into a rigorous *process map*, where the analyst wrote the map (confirming their understanding) and the intermediate-expert validated and expanded it as necessary. Process mapping involves creating a visual representation of a procedure similar to a flow chart, making explicit "the means by which work gets done" [13]. Details of each step (and possible sub-steps) in the process were recorded, such as the duration of each step, other people involved, and possible authoring bottlenecks. The goal of this process map is to make as much of the authoring process as explicit as possible for analysis. In the following case studies, conclusions drawn from the process maps of each team are enumerated as *SSS Components*. We found this process not only helpful, but necessary to discover actionable means by which to improve the authoring experience for the requirements analysis (as the Authoring Support Strategies sections of the case studies will elaborate). It is important to note that it may take multiple of these sessions with the same author (and possibly multiple process maps) to obtain the full authoring process with sufficient detail for analysis. For example, one early process map made with an ABL author was very high-level, focusing on the interconnection with other teams and the bottleneck this caused. The analyst had to return to the ABL author for another session aimed at creating a new process map through the expansion of a single node in the first process map.

3. THE SCENARIO

The scenario we chose is a simplified version of the "Lost Interpreter" scenario recently completed and demoed by the IMMERSE group in ABL [20]. It involves the player as an armed soldier in an occupied territory searching for their lost interpreter via a photograph in their possession. The player must show the image to a cooperative local civilian, who will then recognize 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. The uncooperative civilian will not respond to the player's pleas for help, and if the player is rude or breaks social norms [9], the NPCs (Non-Player Characters) will leave and the scenario will end unsuccessfully.

We chose this scenario because it exercises a wide range of capabilities of interactive characters: player involvement, multiple NPCs with different attitudes, a physical object, communication between NPC and PC, and multiple outcomes. The scenario was also simple enough that each team was able to reach a pseudocode state of completion in a reasonable amount of time (1-3 hours). While the original IMMERSE scenario required non-verbal communication via gesture recognition, we did not enforce that modality 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 extraneous features that all systems may not possess.

4. CASE STUDIES

We studied the following three programmer-author teams of one, two, and five interview participants respectively (although there are more developers on each team). Each architecture made use of a different design philosophy, which will become apparent in the discussion of their individual SSS. The following case studies are listed in order of increasing complexity of the architectures. Each of the case studies provides a description of the authoring process associated with the architecture, a list of the SSS components abstracted from the authoring description, and a description of authoring support strategies that could reduce the authoring burden of the particular SSS.

4.1 Case Study 1: BOD using POSH

Bryson et. al. ascribe to a particular behavior authoring methodology entitled Behavior-Oriented Design [5], an approach that combines Object Oriented Design and Behavior-Based AI [4]. Bryson et al. use BOD and their action selection mechanism, the POSH (Parallel-rooted, Ordered Slip-stack Hierarchical) planner as their architecture and development process because it 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 the scenario was defined, a programmer and designer worked together to create a list of abstract behaviors that need to be performed. It is important to note that the BOD designer (as distinct from the programmer) will never need to encounter anything more complicated than graphical interfaces in their authoring interactions, allowing the designer and programmer to be the most independent of the three case studies (although they may be the same person in some projects). The separation of these two roles is part of the design philosophy of BOD. In our case study the abstract behavior list included seven actions: a greeting/goodbye to mark the beginning and end of the interaction, accepting, examining, returning an item, ignoring the player (for the uncooperative agent), and telling information. The second step was to build what is ultimately a list of procedure signatures for the programmer, determining which of these behavior elements need to be represented as actions and sensors, as well as an idle state should all else fail [6].

The programmer then coded the actions and sensors as functions to create the building blocks of the dynamic plan. In parallel, the designer used the primitives (actions and sensors) created by the programmer to design the behavior tree using ABODE*, a graphical design tool for POSH plans. *Figure 2* shows the BOD/POSH process map for the tasks of the designer and programmer.

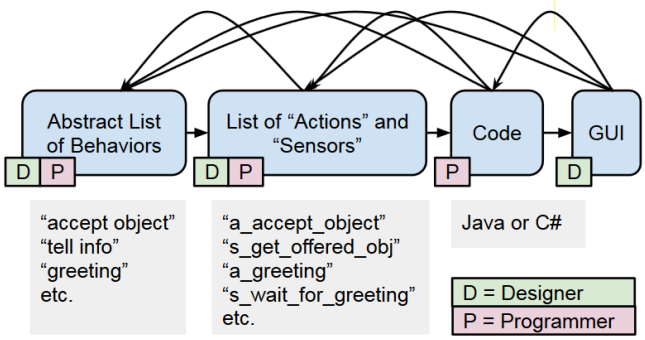

Figure 2: A high-level representation of BOD/POSH's process map including some defined sample primitives

4.1.1 SSS Components

4.1.1.1 Start Minimally

Even though our scenario was relatively simple, it was important to begin with a minimal number of behaviors, actions, and sensors to create a working vertical slice. The scenario began with only four primitives in the first version of the dynamic plan.

4.1.1.2 Decompose Iteratively

A key feature of the BOD authoring methodology is its agility; not only can programmers iteratively tackle the stubs created in the previous Component, but the designer and programmers freely move between phases of the design process to build up missing primitives that were not in the minimal first list. In our case study, the programmer was creating idle and item-handling primitives while the designer realized they had not accounted for the norm-offense response.

4.1.1.3 Minimize and Encapsulate

While not a part of this scenario, an experienced BOD/POSH designer knows that if more than three sensors are needed to trigger a competence, the logic held within the tree is getting too complex. The designer should flag the programmer to offload the logic from the tree onto a new sensor, simplifying the logic (and thus computational resources) controlled by the tree. Not following this rule of thumb is a common mistake most novice BOD/POSH authors make, resulting in a tangled mess of restricting sensors that is difficult to debug and behavior libraries limited to a narrow subset of scenarios. This last SSS Component is the most unique, as all behaviors (which contain the majority of the complexity) are only *triggered* by the tree rather than contained within the tree.

4.1.2 Authoring Support Strategies

The BOD/POSH case study is unique in that it is the only system we studied with an explicit authoring approach as well as a graphical design tool (ABODE*). This makes it easy for novice authors to create simple agents, but authoring and maintaining complex agents creates challenges in need of more robust tools. Thus, the focus of our authoring support strategies will primarily address SSS Component 4.1.1.3, as the first two are well-supported via the BOD methodology and the current architecture.

There is no standardized method for testing and debugging in BOD/POSH, a problem that all the other architectures in this paper also share. Support for syntax checking and live behavior debugging would shorten the programmer's development cycle considerably while iterating on more challenging behaviors (SSS Component 4.1.1.2). Most crucial, however, is a mechanism to facilitate better behavior sharing and reuse between and within projects. The larger a BOD/POSH behavior library is, the more likely that novice users tend to develop their own library instead of reusing existing components. The challenge with three or more sensors triggering a behavior, discussed above in SSS Component 4.1.1.3, is one example of an authoring lesson that needs to be encoded in the graphical design tool to help authors build more reusable behaviors. A new module that manages past similar encapsulated behavior libraries, and prompts users to submit their new simplified behaviors for future reuse, would also increase the reuse and power of BOD and POSH enormously.

4.2 Case Study 2: FAtiMA

FAtiMA [8] is a multiagent architecture in which each agent has an emotional state and plans future actions to achieve a specific goal. Goals can be weighted according to their relative importance. Different characters can have separate personality files in which these weights are defined. Authoring in FAtiMA is done by editing several separate XML files. When presented with the requirements of the the Lost Interpreter scenario, FAtiMA authors started by considering the motivations of the NPCs. Since the behaviors of agents in FAtiMA are goal driven, it was proposed that NPC's in this scenario must have the explicit goal of helping the player. A possible example of such a goal is shown in *Code 1* with the goal *Help*.

Code 1 ActivePursuitGoal

```
<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>
```

Additionally, there needed to be a motivation not to help, in order to model the uncooperative NPC's behavior. The authors chose for the uncooperative NPC to have the goal of avoiding harm from the armed player (let it be called *ProtectSelf*). For this second goal to be useful, there must be an NPC action that is helpful to the player, but at the same time might put it in harms way. For instance, the NPC might consider the possibility of being harmed when taking the picture from the player. If the agent considers a plan involving possibly being harmed, then it will feel a Fear emotion. The authors then continued to define actions that the agents can take along the path of reaching the help goal, such as actually taking the photo, examining it, or speaking.

4.2.1 SSS Components

We were lucky in this case study to consult with a second FAtiMA authoring team after iteratively discovering the SSS Components with the first team. Their responses have been included in the following sections alongside those of the first FAtiMA scenario authoring team.

4.2.1.1 Goals First

FAtiMA's 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 (e.g. respond to request or 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, with the appropriate set of actions, the agent should be able to deal with a wide range of situations, and thus goals. We speculate this different approach may be caused by the disparity between the author experience and scenario complexity between the two teams (the second team had more experience and a less demanding scenario).

4.2.1.2 Find Decision Points

We noticed that the 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, it helped to author the previous SSS Component 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 this analysis. Their modeled scenarios were required to go through sequential phases, due to pedagogical objectives. Thinking of the decision point sequences helped define their goals.

4.2.1.3 Goal Weighting and Tuning

The cooperative and uncooperative civilians in the scenario chose to take different actions when deciding to help by having different numerical weights for the *Help* and *ProtectSelf* goals. By giving more importance to a particular goal in the character's personality file, the interviewed authors made sure each agent made the appropriate decision at the decision points. It is these goal weights that completely control how different agents take different paths throughout the performance, which 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 [2].

4.2.1.4 Intent Goals for Future Consequences

While not part of this particular authoring scenario, we did encounter a useful authoring anecdote that sparked discussion of this additional FAtiMA SSS. Goals have two types: Active Pursuit and Intent. For Active Pursuit goals, the agent actually creates plans to achieve them. Intent goals define constraints that the agent should try to maintain as it pursues Active Pursuit goals. In the process of researching [11], Gomes created two Active Pursuit goals that an agent simultaneously tried to achieve. However, Gomes learned from an expert FAtiMA author that FAtiMA could not handle more than one Active Pursuit goal at a time and had to re-write his entire goal structure. The second FAtiMA team did not agree that this was an important part of their process, as their authors reported easily choosing between Active Pursuit and Intent goals.

4.2.2 Authoring Support Strategies

We propose authoring support strategies for the two SSS Components that were supported by both teams: 4.2.1.2 and 4.2.1.3. For SSS Component 4.2.1.2, we proposed an interface where authors could create example sequences of events schematically. Afterwards, the tool would prompt the user when a given agent was faced with a decision. The author would then create the corresponding goals (and possibly actions) that would motivate different strategies.

All three case studies have points in their authoring where quick iteration of different scenarios would be incredibly helpful in speeding up the authoring process. FAtiMA exhibits the most obvious case, as all of its content adjustments can be narrowed to values in a few specific files. The authors speculated launching multiple simultaneous configurations of a scenario with FAtiMA agents encoded with different personality weights (possibly in real-time), choosing the best version, and iteratively repeating to tune the weights.

4.3 Case Study 3: ABL

ABL was designed with a focus on the creation of expressive IVAs and provides a feature-full reactive planning language for structuring and creating them with a high degree of interactivity [17]. The primary structure primitive in ABL is the behavior, which can subgoal other behaviors in sequence or in parallel and contains preconditions that gate whether or not it can currently be executed. The Active Behavior Tree (ABT) encodes the current intention structure of the agent, with the leaves of the tree as potential 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 that all the interviewed ABL subjects are involved with the in-progress IMMERSE project; we will take care to delineate ABL language-specific and IMMERSE project-specific constructions in this section.

The ABL authors approached the scenario by first creating a list of abstract behaviors which were stubbed into the ABT in a rough sequential structure. At a high level, the authors each tackled a specific behavior and worked iteratively with each other to bring it to completion. ABL authors thus also employ the SSS Components 4.1.1.1 and 4.1.1.2 described above, and so they will not be restated. However, the details of the iterative steps for ABL hold rich opportunities for further SSS Components.

Consider the example give_object() behavior for a character to hand an object to another character:

Code 2 A sequential ABL behavior for requesting an object.

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

The behavior in *Code 2* illustrates the basic behavior structure that authors of abstract behaviors must address:

- *The context of how the behavior will be triggered*: in this scenario, the author knows that give_object() will be triggered in response to a request_object() behavior or it will be accepted unconditionally. It contains no logic for having the offered object rejected. This behavior also only handles removing the object from the character's hand, and assumes another behavior handles the object's fate.

- *Relevant signals and WMEs*: The previous behavior was authored assuming that the characterPhysicalWME contains locational information, that there is a socialSignalWME ready to handle socialInterpretationExtendHand, and that there are constants such as the cExchangeObjectDistance previously defined and calibrated for the world. If any of these are lacking, or the

Table 1: A summary of SSS Components described throughout the paper

Section #	Name	Summary	Systems	Authoring Support
4.1.1.1	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
4.1.1.2	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
4.1.1.3	Minimize and Encapsulate	The BOD/POSH tree relies on simple logic to execute quickly, so complex sensory preconditions should be offloaded to behaviors	BOD/ POSH	A module that manages encapsulated behaviors, keeping them simple and proposing them to new authors
4.2.1.1	Goals First	The agent's actions are driven by goals, so there must always be a goal structure	FAtiMA	Combined with SSS Component 4.2.1.2
4.2.1.2	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
4.2.1.3	Goal Weighting and Tuning	Agent's different behaviors are driven by different weights, which is a huge time sink to debug	FAtiMA	Parallel execution and real-time adjustment/comparison of values
4.2.1.4	Intent Goals for Future Consequences	Language-specific limitations, such as only having one active goal at a time, hinder novice-intermediate authors	FAtiMA	Better documentation
4.3.1.1	Define Coding Idioms	As ABL is its own language, an author must have a strong understanding of their chosen idioms	ABL	Too advanced for a tool to offer much help
4.3.1.2	NPC and Player Considerations	An author must conceptualize roles, the contents of the working memory and ABT, and fine-grain performance details while building up their behaviors	ABL	Revival of the ABL Debugger through modularization: offline code analysis of behavior structures through idioms
4.3.1.3	Consider Interruptions	Authors must try to robustify their behaviors against interruptions and stalling, which complicates the previous SSS Element	ABL	Revival of the ABL Debugger through modularization: offline code analysis of behavior structures through idioms

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 behavior. 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 challenging and crucial step to making these behaviors robust is handling interruptions, which the above behavior fails to do. In the success_test, if the NPC never acknowledges the socialSignal or the player never comes in range, the NPC will hold their hand out forever. If a timeout was added to holding out their hand, what should the NPC do about the unrequited object offering, and how should the lost request_object() context be handled? These are all considerations the author must address when making behaviors robust.

4.3.1 SSS Components

4.3.1.1 Define Coding Idioms

Unlike BOD/POSH and FAtiMA, which make strong architectural commitments to specific agent authoring idioms, ABL is a more general reactive planning language, within which many different idioms can be implemented. Before novice and intermediate programmers can make progress, generally an expert ABL programmer must first define the coding idioms used to structure the agent (see [25] for examples of ABL idioms). These idioms define approaches for organizing clusters of behaviors to achieve goals. For the IMMERSE project, an idiom called Social Interaction Units (SIUs) has been developed to organize clusters of behaviors around goals to achieve specific social interactions. The ABL authors interviewed all approached the "Lost Interpreter" task using the SIU idiom.

4.3.1.2 NPC and Player Considerations

Although we can see that the example behavior 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 personally maintain regarding how the behavior will be triggered in the performance, whether NPC or PC characters will be performing or responding to the behavior, and what supportive information must be stored in working memory. BOD/POSH and FAtiMA offload much of this complexity into the actions

implemented in the game engine, while ABL keeps this complexity within the decision-making process of the agent.

4.3.1.3 Consider Interruptions

In the ABL scenario, if the system detects the player offering the photo, it will trigger the series of ABL behaviors 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), this will trigger the NPC to give back the photo regardless if it is in the middle of another behavior such as pointing. From a designer's perspective, it is perfectly logical that someone may return the photo with one hand and point with another. However, the author of point_to() must account for the fact that the behavior may have to multitask with other behaviors to dynamically decide which hand to use. If the synchronization of those behaviors is not done properly, the animation of the IVA will contain artifacts which are not appealing.

4.3.2 Authoring Support Strategies

We discussed ABL's SSS with novice, intermediate, and expert authors of the ABL language, and their processes all shared the same structure described in detail above. However, novices and early intermediate authors *needed* to reference experts to understand that the above considerations existed, and where to look for them in the code or how to create aspects of them if they were missing. Once example behaviors have been created, authors routinely copy-paste and adapt existing code. ABL meta-behaviors, an advanced language feature, could help alleviate this process, but they were not utilized by any of our authors.

Novice-intermediate ABL authors work within previously established idioms, such as IMMERSE's SIUs. Making tools to support the design of new idioms in regard to SSS Component 4.3.1.1 is not within the scope of this approach, as we have been focusing primarily on novice-intermediate authoring tool support.

In contrast to the visual representation of BOD/POSH's dynamic plan, the Active Behavior Tree (ABT) in ABL is in a constant flux, making it hard to visualize. Currently, ABL authors use debug log print statements of the current system state and trial-and-error experiments to understand ABT dynamics. Support for more sophisticated debugging techniques does exist in the form of an ABL debugger (a process that executes alongside an agent at runtime), but none of the ABL authors choose to use it. The current ABL debugger contains too many usage barriers to ascertain if it is technically useful in helping visualize SSS Components ix and x. Our current plan for overcoming these usage barriers include a graphical ABT representation that allows for parallel viewing of disjoint parts of the tree, saving tree snapshots, and saving viewing locations for repeated tests. We also have plans to analyze ABL code structure offline through an IDE plugin, and an ABT pattern recognition algorithm to alert authors to missing behavior cases (NPC vs. PC implementations), unused behaviors, and other structural indicators we have yet to find.

5. DISCUSSION

The SSS Components that arose from the simplest case study, BOD/POSH, were high-level authoring guidelines that apply to multiple architectures. Specifically, all three of BOD/POSH's Components apply to ABL as well, as they are more characteristic of a hierarchical planning structure than of BOD/POSH specifically. Other SSS Components, such as FAtiMA's *Goals First* (iv), are guided by FAtiMA's planning-oriented cognitive-appraisal architecture that is driven by explicit goals. The ABL case study provides a level of complexity above the other two; ABL is a general reactive planning language where many authoring idioms may be designed, as well as the only architecture in the case studies with a behavior tree that dynamically changes during runtime.

Many interviewees were resistant to the idea of specifying implementation time (in number of hours), as it varied greatly between each task. We also found that the particular shape or contents of any single process map wasn't as relevant as the process of elucidation and reflection. The goal of the process mapping technique is to tease out what is general and what is system-specific about a given architecture. The system-specific information forms the core of requirements analyses and the actionable plans found therein. *Table 1* shows a summary and consolidation of each team's analyst's best attempt at discovering system-specific patterns of frustration and proposing solutions to alleviate the problems.

Although the SSS concept contains the phrase "System-Specific" in its name, we found that certain SSS Components are shared between different systems, revealing common architectural tropes. However, we did find common medium-level authoring challenges that may be of use to other teams by abstracting SSS Components of the case studies: the need for (better) mechanisms for behavior (or other architecture construct) sharing and reuse, live debugging, and template structures for architecture constructs. We hope that the SSS Components defined in this paper not only help other architectures discover their own SSS Components, but that the other architectures can reuse the SSS Components and the corresponding authoring support strategies we have outlined.

6. CONCLUSIONS

This paper has proposed the SSS requirements methodology as a means by which programmer-authors may better understand their IVA architecture's authoring burden and make progress toward alleviating that burden. The methodology was born of interviews conducted with many disparate and independent groups performing IVA authoring research. We then performed case studies of three teams authoring a single simple scenario where we process-mapped their authoring process, extracted and elaborated their SSS and its Components, and proposed authoring strategies that might alleviate their authoring burdens. The three teams found the SSS to be a valuable tool in analyzing their system, and each group plans on implementing support for their authoring strategies.

7. ACKNOWLEDGEMENTS

In no particular order: Martin van Velsen, CTAT, Claudio Pedica, CADIA, Samuel Mascarenhas, GAIPS, Andrew Stern, Dan Shapiro, the IMMERSE team, the AmonI group, Megan Pycroft, and Kerry Bruce

8. REFERENCES

[1] Becroft, D., Bassett, J., Mejía, A., Rich, C., & Sidner, C. L. 2011. AIPaint: A Sketch-Based Behavior Tree Authoring Tool. In *AIIDE '11*.

[2] Bernardini, S., & Porayska-Pomsta, K. (2013). Planning-Based Social Partners for Children with Autism. In *Proc. of the Twenty Third International Conference on Automated Planning and Scheduling (ICAPS-13)*.

[3] Bickmore, T., Bukhari, L., Vardoulakis, L. P., Paasche-Orlow, M., & Shanahan, C. (2012). Hospital buddy: A persistent emotional support companion agent for hospital patients. In *Intelligent Virtual Agents* (pp. 492-495). Springer Berlin Heidelberg.

[4] Brooks, R. A. (1986). A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, *2*(1), 14-23.

[5] Bryson, J. J. (2001). *Intelligence by design: principles of modularity and coordination for engineering complex adaptive agents* (Doctoral dissertation, Massachusetts Institute of Technology).

[6] Bryson, J. J., & Stein, L. A. (2001). Modularity and design in reactive intelligence. In *International Joint Conference on Artificial Intelligence* (Vol. 17, No. 1, pp. 1115-1120). Morgan Kaufmann.

[7] Champandard, A. J. (2003). *AI game development: Synthetic creatures with learning and reactive behaviors*. New Riders.

[8] Dias, J., Mascarenhas, S., & Paiva, A. (2011). Fatima modular: Towards an agent architecture with a generic appraisal framework. In *Proceedings of the International Workshop on Standards for Emotion Modeling*.

[9] Evans, Richard (forthcoming). Computer Models of Social Practices. In Vincent Muller (ed.), *Synthese Library: Philosophy and Theory of Artificial Intelligence*. Synthese.

[10] Gaudl, S., Davies, S. and Bryson, J. J., 2013. Behaviour oriented design for real-time-strategy games: An approach on iterative development for STARCRAFT AI. *In*: *Foundations of Digital Games Conference 2013 (FDG 2013)*. Foundations of Digital Games, pp. 198-205.

[11] Gomes, P., & Jhala, A. (2013). AI Authoring for Virtual Characters in Conflict. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[12] Gratch, J., Wang, N., Gerten, J., Fast, E., & Duffy, R. (2007). Creating rapport with virtual agents. In *Intelligent Virtual Agents* (pp. 125-138). Springer Berlin Heidelberg.

[13] Madison, D. (2005). *Process mapping, process improvement, and process management: a practical guide for enhancing work and information flow*. Paton Professional.

[14] Mascarenhas, S. F., Silva, A., Paiva, A., Aylett, R., Kistler, F., André, E., ... & Kappas, A. (2013, May). Traveller: an intercultural training system with intelligent agents. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems* (pp. 1387-1388).

[15] Mateas, M. (1999). *An Oz-centric review of interactive drama and believable agents* (pp. 297-328). Springer Berlin Heidelberg.

[16] Mateas, M. (2003). Expressive AI: Games and Artificial Intelligence. In *DIGRA Conf.*.

[17] Mateas, M., & Stern, A. (2002). A behavior language for story-based believable agents. *IEEE Intelligent Systems*, *17*(4), 39-47.

[18] Nelson, M. J., & Mateas, M. (2009, April). A requirements analysis for videogame design support tools. In *Proceedings of the 4th International Conference on Foundations of Digital Games* (pp. 137-144). ACM.

[19] A. Paiva. GAIPS: Intelligent agents and synthetic characters, 2014. http://gaips.inesc-id.pt/gaips/

[20] Shapiro, D., McCoy, J., Grow, A., Samuel, B., Stern, A., Swanson, R., ... & Mateas, M. (2013). Creating Playable Social Experiences through Whole-Body Interaction with Virtual Characters. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[21] Spierling, U., & Szilas, N. (2009). Authoring issues beyond tools. In *Interactive Storytelling* (pp. 50-61). Springer Berlin Heidelberg.

[22] Carnegie Mellon University. CTAT: Cognitive Tutor Authoring Tools, 2014. http://ctat.pact.cs.cmu.edu/

[23] Reykjavik University. CADIA: Center for Analysis and Design of Intelligent Agents, 2014. http://cadia.ru.is/

[24] Van Velsen, M. (2008). Narratoria, an Authoring Suite for Digital Interactive Narrative. In *FLAIRS Conference* (pp. 394-395).

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