Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2025)

A Method to the Machine: An Architecture for Argument-Driven, Dynamic Character Performance

Kyle Mitchell[1], **Joshua McCoy**[2]
University of California, Davis
1 Shields Avenue
Davis, California 95616-5270 USA
[1]kdmitch@ucdavis.edu [2]jamccoy@ucdavis.edu

Abstract

The evolution of architectures for virtual agents reveals a history of principled trade-offs. While dominant paradigms have successfully optimized for scalability and robustness, other valuable agent characteristics—such as the legibility of intent and the capacity for expressive performance—were often de-prioritized. This paper investigates an alternative set of architectural choices aimed at reclaiming these qualities. We introduce a novel agent architecture inspired by Stanislavskian acting, which operationalizes performance techniques by having agents reason from a core "supertask" and "ask questions" about their circumstances. The computational framework is composed of three layers: a strategic layer for character-centric decision-making, a defeasible logic programming (DELP) reasoning layer for argumentation, and a tactical layer using A Behavior Language (ABL). This paper provides a detailed architectural analysis of the system, demonstrated through a case study in a Unity-based game scenario. We conclude by discussing the system's limitations and engineering challenges, outlining a path for future work.

Introduction

The evolution of architectures for virtual agents reveals a history of principled trade-offs. To move beyond simple, reactive scripts and manage the immense complexity of modern interactive worlds, dominant paradigms have successfully optimized for qualities like scalability and robustness. In making this necessary trade, however, other valuable agent characteristics—such as the nuance of the decision-making process, the legibility of an agent's intent, and its capacity for expressive performance—were often deprioritized. While this approach is effective for many genres, it leaves a crucial gap for certain kinds of experiences, particularly those centered on rich narrative and character interaction. This paper, therefore, investigates an alternative set of architectural choices, exploring whether it is possible to reclaim those expressive and explanatory qualities by synthesizing formalisms from performance theory and symbolic AI.

To successfully reclaim these qualities, our work must address several core research questions. First, what formalisms can we draw upon to model the nuances of believable, character-driven performance? Second, how can we ensure an agent's actions remain coherent with its high-level goals and personality, even when granted the flexibility to react to novel situations? Finally, what architectural approach allows for the deliberative transparency of a more traditionally authored character, while mitigating the prohibitive authorial burden traditionally required to produce such content?

One answer to these questions is grounded in performance theory, specifically Stanislavski's method, a framework centered on the actor's process of "experiencing the role" to achieve an authentic performance. This method posits that a character's journey is unified by a supertask (also known as a super-objective), an objective that operates at the scale of the drama as a whole. This journey is punctuated by a sequence of smaller tasks or obstacles to overcome—which our system models as "behaviors"—that are chosen based on the character's context. The core of this contextual reasoning is captured by Stanislavski's "Magic If." To pursue their supertask authentically, an actor does not simply perform actions, but first asks a "what if" question about their given circumstances. This query process leads to justification: the actor's internal, character-driven reason for taking an action. Our architecture computationally models this process: the supertask provides the high-level goal, the "Magic If" is modeled as a formal query against the agent's knowledge base, and the resulting behavior is the justified, performative outcome.

This paper explores the design and implementation of a novel agent architecture built on these principles. We present a system that integrates this Stanislavskian motivational framework with an argumentation-based logic for deliberative reasoning and a dynamic behavior execution engine for tactical flexibility. While the conceptual basis for this system has been introduced in prior work (Mitchell and McCoy 2024), this paper aims to fill the gap in the literature by providing its first detailed technical description and engineering analysis. Our primary contribution is the in-depth presentation of this system's design, demonstrated through an illustrative case study and followed by a discussion of its architectural implications, limitations, and avenues for future work.

Related Works

A central challenge in creating intelligent virtual agents is the selection of an architecture for authoring and executing

their behaviors. The approaches to this problem range from simple reactive systems to complex, dynamic languages, each with distinct trade-offs in terms of authoring complexity, runtime flexibility, and expressive power. To situate the architecture discussed in this paper, we analyze prior work across three key domains: action, mind, and performance.

Architectures of Action

The architectures governing agent action have evolved from simple, prescriptive models toward systems that afford greater flexibility. Foundational models like the Finite State Machine (FSM) are famously ill-suited for complex characters due to scalability issues (Sekhavat 2017), leading to the rise of more modular and robust solutions. The modern industry standard, Behavior Trees (BTs), offer a powerful, hierarchical control structure (Colledanchise and Ögren 2018), while declarative paradigms like Goal-Oriented Action Planning (GOAP) provide greater runtime flexibility (Orkin 2006). While powerful, these approaches can still present challenges for deep runtime adaptability and authoring nuanced, expressive performances.

Other notable paradigms for complex behavior generation include Hierarchical Task Networks (HTNs) for structured task decomposition (Erol, Hendler, and Nau 1994), data-driven methods like Reinforcement Learning (RL) that optimize for a reward signal (Sutton and Barto 2018), and recent Large Language Model (LLM)-based approaches that can generate behavioral plans from natural language prompts (Wang et al. 2024).

Bridging the gap between structured proceduralism and dynamic planning are specialized programming languages, such as TED (Horswill and Hill 2024) and Step (Horswill 2022). Among these, one of the most influential for believable characters is A Behavior Language (ABL), developed for the interactive drama *Fa-çade* (Mateas and Stern 2004). The execution layer of our architecture uses ABL directly, employing its dynamic capabilities as the substrate for our higher-level reasoning components.

Architectures of Mind

Moving beyond the mechanics of how an agent acts, this section investigates the architectures that determine why an agent chooses to act. The symbolic AI tradition offers a rich history of such models, providing context for our use of defeasible logic. Foundational paradigms for rational agents include comprehensive cognitive architectures like SOAR, which models all intelligent activity as a search through a problem space and features an impasse-driven subgoaling mechanism for reasoning and learning (Laird, Newell, and Rosenbloom 1987), as well as the Belief-Desire-Intention (BDI) model (Bratman 1987). The seminal Procedural Reasoning System (PRS) established BDI as an industrial-strength framework for building rational agents in complex environments (Georgeff and Lansky 1987). To model the messy, non-monotonic reasoning required for more believable characters, our architecture instead uses Defeasible Logic Programming (DeLP). DeLP is an argumentation-based formalism that resolves conflicts between competing rules via a dialectical process, providing a strong computational analogue for a character's internal justification process (García and Simari 2004).

Other symbolic AI approaches have focused on rich knowledge modeling for social and narrative contexts. These range from foundational work like Script Theory (Schank and Abelson 1977), to sophisticated social simulation engines like Comme il Faut (CiF) (McCoy et al. 2011) and the engine used in *Versu* (Evans and Short 2014). The FAtiMA architecture, for instance, uses a double appraisal cycle for socially aware characters (Dias, Mascarenhas, and Paiva 2014), while early work by El-Nasr et al. adapted similar theatrical techniques to ours into the Flame architecture for interactive drama (El-Nasr, Yen, and Ioerger 2000). A parallel thread in narrative planning uses AI planning to generate causally coherent stories where character actions are intentionally motivated (Ware and Young 2014; Ware and Siler 2021).

These approaches highlight a distinction between systems focused on the formal process of reasoning (e.g., BDI, DeLP) and the authored content that informs that reasoning (e.g., scripts, social models). More recently, Large Language Model (LLM)-based "agentic AI" systems have demonstrated impressive capabilities for commonsense reasoning and emergent behavior from natural language prompts. Our work deliberately chooses a symbolic, "glass box" approach, prioritizing the deliberative transparency and inspectability of the reasoning process over the powerful but often opaque nature of these neural architectures. The dialectical process of DeLP is particularly well-suited to our goals, providing a computational analogue for the internal conflict and justification central to the Stanislavskian method.

Architectures of Performance

The ultimate measure of an agent in an interactive narrative is not the elegance of its internal algorithms but the quality of its external performance. A successful agent must perform in a way that is expressive, characterful, and legible to a human user. A critical theoretical lens for this is the concept of Expressive AI, which focuses on creating artifacts perceived as intelligent and meaningful by a human audience (Mateas 2001). This perspective gives rise to the practical goal of "procedural acting," where the execution of a task is imbued with character and style.

Two landmark games that likely many are familiar with, *Façade* and *The Sims*, serve as canonical and contrasting case studies in this pursuit. *Façade* is a one-act interactive drama driven by the AI performances of its two main characters, using ABL to manage behavior and maintain a coherent dramatic arc (Mateas and Stern 2003). In stark contrast, *The Sims* is a life simulator that generates emergent behavior from the bottom up using a "smart objects" architecture where agents are driven by an internal, utility-based needs system.

The comparison between these systems illuminates a central philosophical dialectic in interactive narrative: the tension between authorial control ("strong story") and agent agency ("strong autonomy"). Our architecture that combines a language from the *Façade* lineage (ABL) with an au-

tonomous internal reasoning engine (DeLP) can be seen as an attempt to synthesize these two opposing poles. Other architectures have also sought to bridge this authoring gap. Thespian, for example, introduces an "actor-centric" model where a "fitting" algorithm automatically tunes goal-driven agents to match author-provided scripts, allowing for both adherence to authored scenarios and robust, in-character improvisation when a user deviates (Si, Marsella, and Pynadath 2005). Similarly, early work by Cavazza et al. on immersive storytelling used a motivation-based planning system to allow a user to step into the role of a character and influence the story's performance in real-time (Cavazza et al. 2007). More recently, the Puppitor system demonstrates a similar decoupling of character and persona to our own, using a performance-centric system to drive low-level animation behaviors (Junius et al. 2022).

A contrasting, data-driven approach to performance shifts the focus from authoring agent behaviors to generating the possibility space for performance itself. Some work on automated game design, for instance, uses machine learning to learn from and recombine the core concepts of existing games to create entirely new ones (Guzdial and Riedl 2018). This reframes the architecture of performance not as the logic controlling a single character, but as the generative system that creates novel, playable experiences by learning underlying design patterns.

This ambition has deep roots in AI research, dating back to foundational systems like TALE-SPIN, which demonstrated that a narrative could be generated from a simulation of a character's reasoning process (Meehan 1977). Our architecture builds on this long tradition, using ABL as the sophisticated "body" of the agent and DeLP as the "mind" for deliberation.

System Architecture

Our system is composed of three distinct architectural layers, implemented across different environments to leverage the strengths of each. At the highest level is the strategic layer, written in C# as a set of components within the Unity game engine. This component embodies the Stanislavskian framework, manages the agent's *Supertask*, and contains the core selection algorithm that interrogates potential lines of reasoning to choose the most character-appropriate behavior. This strategic layer is served by the second component, a reasoning layer which utilizes a Java-based Defeasible Logic Programming (DeLP) package (García and Simari 2004) running on an external server. The role of DELP is to generate multiple, justifiable arguments for various courses of action based on the agent's beliefs. Finally, the decision from the strategic layer is passed to the third component: a tactical execution layer built using A Behavior Language (ABL) (Mateas and Stern 2002) and co-located with the reasoning layer on the Java server. This ABL layer enacts the chosen plan by dynamically spawning behaviors onto a runtime-managed behavior tree. This section will now detail the design and engineering of each of these three layers. A complete system diagram illustrating their interaction is shown in Figure 1.

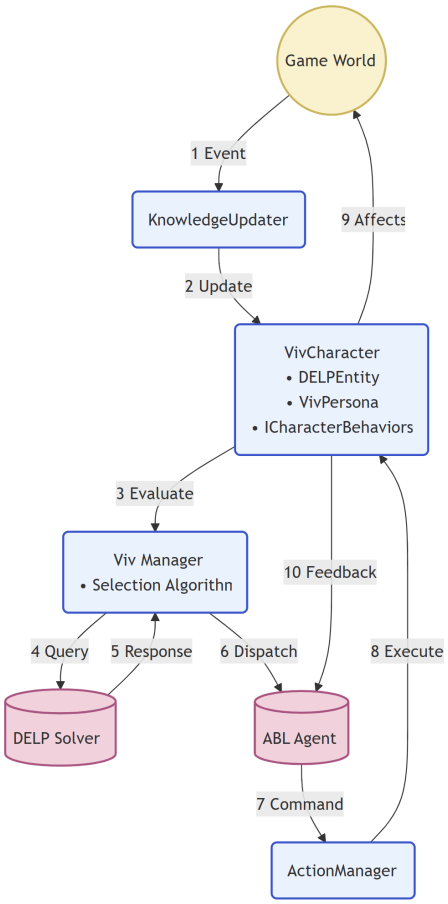

Figure 1: The diagram illustrates the three primary layers and the flow of information between components. Core Viv system components (blue, rounded rectangles) and the Game World (yellow, circle) run in the Unity client. The reasoning (DELP) and tactical (ABL) servers (pink, cylinders) run as an external Java process. The numbered arrows trace a complete decision cycle, from a world event (1) to the final character action (9) and the feedback loop that closes the cycle (10).

Viv: The Strategic Layer

As established in previous work (Mitchell and McCoy 2024), the architecture's strategic layer is grounded in Stanislavski's method to achieve a more authentic and intelligible performance. This foundation provides the agent with a supertask—a primary, unifying objective for the character's journey—and a process of "asking questions" about its given circumstances. In this system, these questions manifest as a set of potential *Behaviors*: actions that could reasonably help the character achieve its *Supertask*. The core of the Stanislavskian "Magic If" is computationally modeled as a query process: each *Behavior* is defined by a set of underlying *Assumptions* which are validated against the agent's knowledge base. The relationships between the core components of this layer are illustrated in Fig. 2.

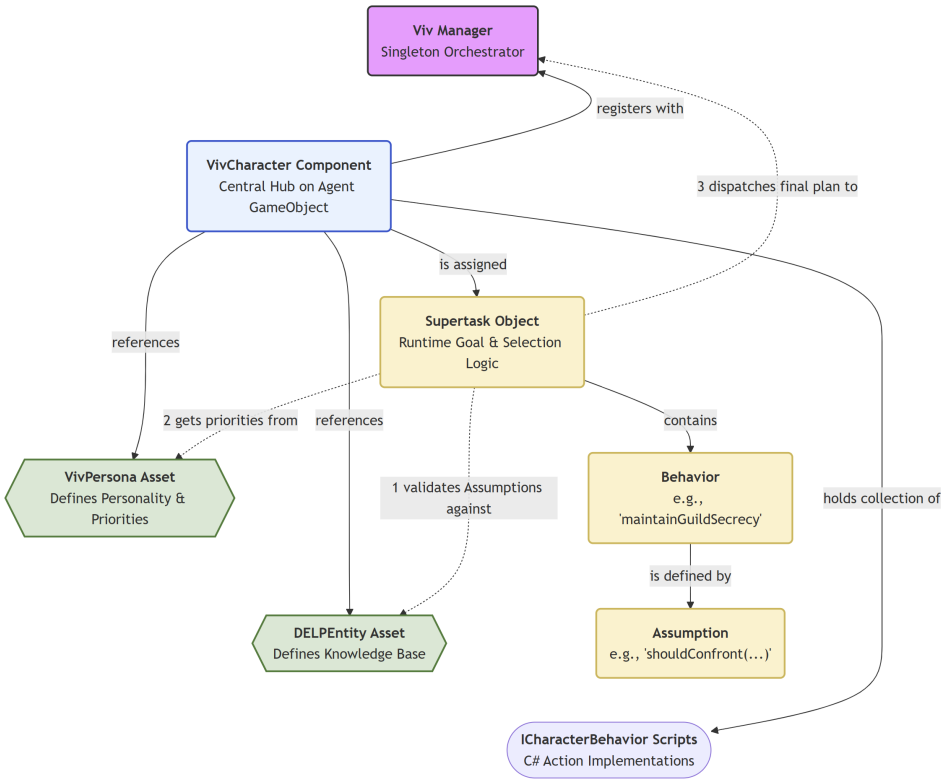

Figure 2: This diagram illustrates the relationships between the core components described in Section 3.1. The *VivCharacter* component acts as a central hub, holding references to configurable ScriptableObject assets like the *VivPersona* and *DELPEntity* (solid lines). Dotted lines show the functional data flow during the selection process, where the runtime *Supertask* object queries these assets to make a decision and dispatches the final plan to the *Viv Manager*. Different shapes denote MonoBehaviour components (rounded rectangles), ScriptableObject assets (hexagons), and other runtime objects or data structures.

The *Viv* Manager Component The central orchestrator of the strategic layer is the *Viv* component, a singleton manager in the Unity scene. Its primary functions are scene-wide service provision and management. The *Viv* component maintains a static registry of all active *VivCharacter* instances and acts as a factory for *Supertask* objects. Crucially, after its internal selection algorithm has chosen a behavioral plan, the *Viv* manager handles the serialization of that plan and dispatches it as a message over its TCP/IP connection to the external ABL server.

The *VivCharacter* Component Individual agents are represented by GameObjects equipped with a *VivCharacter* component. This component acts as the primary data hub for an agent, holding references to its unique *DELPEntity* knowledge base, its *VivPersona* asset, its currently assigned *Supertask*, and a collection of *ICharacterBehavior* implementations. These implementations are snippets of C# code that allow the character to act upon commands ultimately received from the ABL server. Each *VivCharacter* registers itself with the central *Viv* manager and determines its own initial *Supertask* to request.

The *VivPersona* Asset A character's personality is defined by a *VivPersona*, which is implemented as a ScriptableObject asset in Unity. This design allows personas to be created, edited, and reused as assets directly within the editor. Each *VivPersona* asset contains a list of *BehaviorPriority* objects, which map a *Behavior* name to an integer priority score. This provides a straightforward, designer-friendly way to specify a character's behavioral inclinations.

The *Supertask* Object and Selection Process A *Supertask* is a C# class instance that represents a character's active, high-level goal and contains the full decision-making logic. The process begins with validating the *Assumptions* for all its associated *Behaviors*. The validation of an *Assumption* is an asynchronous, event-driven process that queries the external DELP reasoning layer. When a *Supertask* begins its evaluation, each of its *Assumptions* registers as a listener with a global EventManager and sends its resolved query string (e.g., isHostile(player)) as a message to the DELP server. The *Assumption* then waits until the EventManager broadcasts a matching DELPResponse event. Upon receiving its specific response, the *Assumption* updates its validity status (e.g., to YES, NO, or UNDECIDED), unsubscribes from the event, and notifies its parent *Supertask* that it has completed.

Once all pending *Assumption* responses have been received, the *Supertask*'s selection algorithm proceeds as de-

Algorithm 1: Supertask Behavior Selection

```
function SelectBehaviors(allBehaviors, persona)
finalPlan ← []

// Tiered validity check is encapsulated in this function call
candidateBehaviors ← GetCandidateBehaviors(allBehaviors)

// Select final plan from the pool of candidates
compatibleSets ← FindAllCompatibleSets(candidateBehaviors)
if compatibleSets is not empty then
  finalPlan ← GetLargestSet(compatibleSets)
else if candidateBehaviors is not empty then
  SortBehaviorsByPriority(candidateBehaviors, persona)
  finalPlan ← [candidateBehaviors[0]]
end if

return finalPlan
```

tailed in Algorithm 1. This process first constructs a pool of candidate behaviors using a tiered validity check: it prioritizes any behavior built on only "YES" assumptions, but will fall back to a pool including "UNDECIDED" assumptions if no purely valid options exist. Any behavior with a "NO" assumption is always discarded. From this final candidate pool, the algorithm then determines the final plan based on compatibility and persona priority. This interaction, along with the validation of *Assumptions* against the *DELPEntity*, is represented by the dotted and numbered functional arrows in our component architecture diagram (Fig. 2).

This entire selection process, from query to dispatch, is designed to computationally model an actor's character-building process. The validation of an *Assumption* is a direct analogue for Stanislavski's "Magic If": the agent poses a "what if" question to its understanding of the world (the DELP knowledge base) and awaits an answer. The selection algorithm then models the character's reaction to that answer, filtered through the lens of its innate personality (*VivPersona*). Furthermore, this component-based architecture facilitates extensibility. By separating knowledge (*DELPEntity*), personality (*VivPersona*), and actions into distinct, interchangeable assets, designers can author a wide range of character archetypes directly within the Unity editor, often without changing the underlying code.

DeLP: The Reasoning Layer

The strategic layer is served by a reasoning layer that provides the formal justification for an agent's actions. This layer is implemented as an external Java application that runs a DeLP solver based on the Tweety Project libraries for computational models of argument (Thimm 2017). It communicates with the Unity-based strategic layer via TCP/IP, receiving updates to an agent's knowledge base and responding to queries. This separation allows the computationally intensive process of logical deliberation to occur in a dedicated environment.

The *DELPEntity*: An Agent's Knowledge Base An agent's entire "worldview" or knowledge base is encapsulated within a *DELPEntity*, which is implemented as a ScriptableObject asset in Unity. This design choice is critical for authoring, as it allows designers to create, reuse, and assign complex knowledge bases to different characters directly in the editor. Each *DELPEntity* asset contains three lists of strings that represent the core components of a DeLP program: Facts, Strict Rules, and Defeasible Rules. This class acts as the primary interface in Unity for managing an agent's knowledge base. To facilitate runtime changes, the *DELPEntity* class exposes a simple API with methods such as AddFact(string), RemoveFact(string), AddStrictRule(string), and AddDefeasibleRule(string). Any call to these methods automatically triggers a synchronization process that serializes the agent's complete set of facts and rules into *DELPMessage* objects—data structures defined for network communication—and sends them to the external Java server.

The Argumentation and Query Process As described in the previous section, the validation of a character's *Assumption* is what triggers a query to this reasoning layer. This process creates a *DELPMessage* with a specific query code and sends the resolved query string to the server. The external DeLP server then performs its core function: a dialectical analysis. It attempts to build a logical argument for the query and pits it against all possible counter-arguments derived from the agent's full knowledge base (García and Simari 2004). The server's final judgment is returned as a *DELPResponse* message containing the answer "YES," "NO," or "UNDECIDED," which then informs the strategic layer.

Dynamic Knowledge Updates To ensure an agent's knowledge base remains current with the evolving game state, the architecture includes a dynamic update system managed by a singleton *KnowledgeUpdater* component. This system uses *KnowledgeUpdateRule* ScriptableObject assets to link game events to changes in an agent's knowledge. An example of such a rule is shown in Fig. 3. Each rule defines an interactionType and outcome and specifies a factToAdd or factToRemove. The *KnowledgeUpdater* listens for global game events and, upon finding a matching rule, determines which characters are affected based on RuleApplicability criteria (e.g., DirectParticipantsOnly, AllCharacters). It then processes the rule's fact template string and calls the appropriate API method on that character's *DELPEntity*.

This event-driven, asset-based system was designed to support a large domain of gameplay interactions. The workflow for adding a new "event-triggers-fact" relationship is significantly streamlined: a designer can (1) define a new outcome string in a shared constants file, (2) create a new *KnowledgeUpdateRule* asset in the Unity editor, (3) configure the asset's fields to link the event strings to the desired fact change and applicability, and finally, (4) add a single line of code in the relevant gameplay system to invoke the global interaction event. This decoupling of game logic from

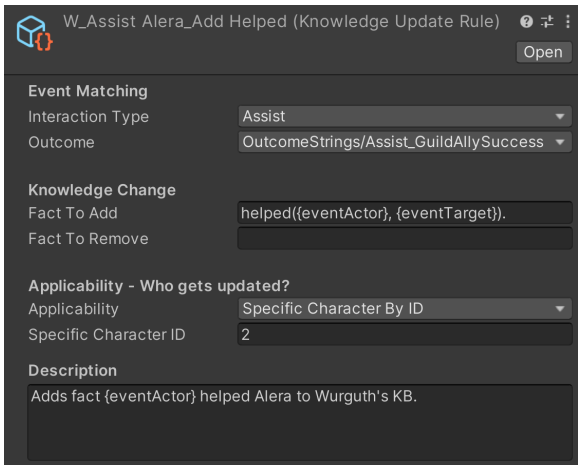

Figure 3: This ScriptableObject allows a designer to author logical reactions to game events without much code. The designer specifies an Event Match (interactionType, outcome), a resulting Knowledge Change (factToAdd), and the rule's Applicability (e.g., who learns the new fact), which decouples narrative logic from gameplay implementation.

knowledge update logic facilitates rapid development and makes the system highly maintainable.

ABL: The Tactical Layer

The tactical layer is responsible for the concrete, moment-to-moment execution of the behavioral plan selected by the strategic layer. This component is implemented as a single ABL GameAgent that runs on an external Java server. It employs what can be described as a "hivemind-daemon" pattern; this architecture was a deliberate design choice aimed at reducing authorial burden. By defining a single, shared library of tactical ABL behaviors (e.g., moveToTarget), all Viv-controlled characters can draw from the same set of capabilities. This approach reduces the need to create and maintain a separate ABL agent program for each individual character, while remaining extensible enough to support character-specific behaviors if the need arises. A sequence diagram shown in Fig. 4 illustrates the full communication loop between this layer and the strategic layer.

From Strategy to Tactics: The *VivWME* and Goal Spawning The communication from the strategic layer to the tactical layer is encapsulated in a *VivWME* (Working Memory Element). When the *Viv* manager in Unity selects a plan, it creates a *VivWME* instance, populates it with an array of *SpawnGoalData* objects, and sends it to the ABL server. On the ABL side, a persistent lookForVivCommands behavior continuously waits for a new *VivWME*. Upon detecting one, it iterates through the goals in the toSpawn array and uses ABL's spawngoal command to initiate the corresponding high-level tactical behavior within a single, parallel "hivemind" behavior.

From Tactics to Action: Primitive Acts and Status Feedback The high-level tactical behaviors are authored in ABL as sequences of primitive acts, which correspond to concrete actions in Unity. When the ABL agent executes an act, it sends a command message to the Unity client and then immediately pauses its own execution using a wait step with a success test. This test is designed to wait for a specific *BehaviorStatusWME* to be sent back from Unity, signaling that the low-level action has completed. Only when the correct *BehaviorStatusWME* is received does the tactical behavior's wait condition resolve, allowing it to proceed to the next step in its sequence.

The Unity Execution Bridge The command message sent by an ABL act is received in Unity by the singleton *ActionManager*. This manager uses a factory pattern (*ABLActionFactory*) to map the incoming command name to a specific *IABLAction* object. This object's Execute method parses the command's data and calls the appropriate public method on the target *VivCharacter*, adding the command to the character's private queue. The *VivCharacter*'s Update loop manages the lifecycle of the corresponding *ICharacterBehavior* script (Enter, Update, Exit). Upon completion, the controller instantiates a new *BehaviorStatusWME* with the result of the action and sends it back to the ABL server, satisfying the wait condition and allowing the tactical plan to advance.

The authoring process for extending this tactical library with a new behavior, while meticulous, is well-defined. A developer must define the new sequential behavior and its primitive acts in the ABL program and implement the corresponding Java action classes on the server. On the Unity client, a new *IABLAction* class must be written to parse the command and be registered in the *ABLActionFactory*. Finally, the corresponding *ICharacterBehavior* component must be implemented and attached to the *VivCharacter* GameObject. While this process involves multiple touchpoints across the client-server divide, it ensures that every action is explicitly defined, from high-level tactical sequencing down to low-level engine execution.

Case Study: The Guild of Shadows

To demonstrate how the architectural components work in concert to produce dynamic, context-aware behavior, this section presents a case study. The case study revolves around a scenario wherein the player is tasked with investigating a secretive organization, the Guild of Shadows. We will focus on the behavior of a single non-player character (NPC), Wurguth, whose primary role is to act as an enforcer for this guild. To showcase the system's ability to generate different outcomes from multiple lines of reasoning, we will trace Wurguth's decision-making process through two distinct pathways in this scenario, each dictated by different player choices.

Wurguth's Initial State and Persona

At the start of the scenario, Wurguth is assigned a single, high-level *Supertask*: ProtectGuildOfShadows. This *Supertask* is associated with a set of four potential *Behaviors* that represent his primary tactical options. His character is defined by his *VivPersona* asset, which establishes a priority

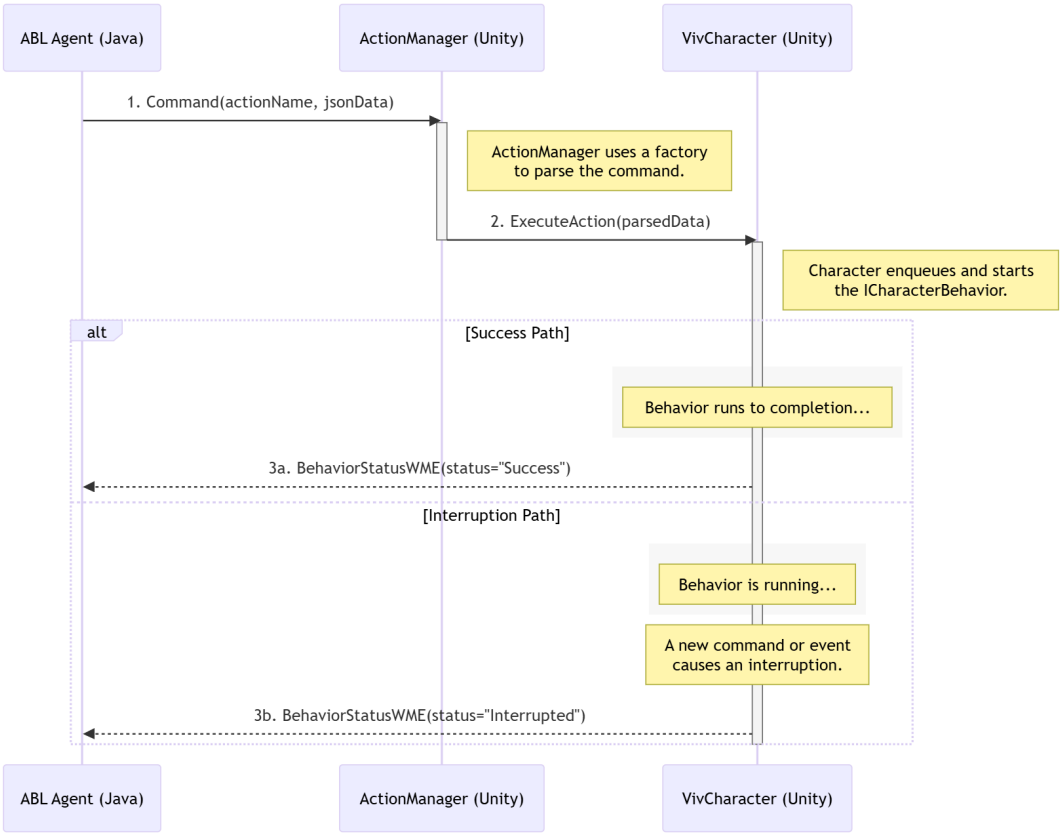

Figure 4: This diagram shows the interaction between the external ABL Agent (Java) and core Unity components. A primitive act command from ABL (1) is received by the ActionManager and dispatched to the VivCharacter (2). The character executes the behavior over multiple frames. The alt block illustrates the two primary outcomes for this final step: the behavior can run to completion and send a "Success" status back to the ABL agent (3a), or it can be preempted by another action and send an "Interrupted" status (3b). In either case, this feedback closes the loop and allows the ABL plan to either proceed, or terminate.

order for these *Behaviors* to reflect a cautious but loyal nature: *investigateSuspiciousActivity* is given the highest priority, followed by *gatherInformation*, *maintainGuildSecrecy*, and finally *neutralizeThreat* as the lowest priority. His initial *DELPEntity* knowledge base contains the facts and rules relevant to his role. This includes key facts about guild membership, a strict rule defining that guild members are allies, and several defeasible rules that govern when he should become suspicious or consider confronting a target.

Pathway 1: Friend of My Friend

The first pathway begins when the player chooses to help Alera, Wurguth's previously established guildmate, with her quest. Upon completion of this task, the game's quest logic fires a global event. This event is detected by the *KnowledgeUpdater* component, which finds a matching *KnowledgeUpdateRule* asset and updates Wurguth's *DELPEntity* by adding the new fact: $helped(player, alera)$.

The addition of this new fact triggers an immediate re-evaluation of Wurguth's *Supertask*. The reasoning layer is queried, and the new fact allows it to build a new, warranted argument. As illustrated in the diagram (Fig. 5, Pathway 1), the system uses this new fact and its existing rules to justify the conclusion $shouldConfrontPlayer(wurguth)$. This logical chain represents Wurguth's social reasoning as authored by the designer; the rules model the social heuristic that an individual who helps an ally is likely not a threat, justifying a non-hostile response.

With this new information, the strategic layer's selection algorithm evaluates Wurguth's valid *Behaviors*. The two primary valid options are now *gatherInformation* (whose prerequisite $\neg isSuspiciousOf$ is met) and *maintainGuildSecrecy* (whose prerequisite *shouldConfrontPlayer* is also true). Consulting the *VivPersona* priorities and the behavior compatibility list, the algorithm determines that both of these non-hostile behaviors are valid and compatible. The *Viv* manager therefore dispatches a *VivWME* containing both *Behaviors* to the ABL server for parallel execution. On the server, the ABL agent begins executing the high-level *maintainGuildSecrecy* tactic. This ABL behavior, in turn, uses a primitive act command for CalmConfrontation, sending a message back to the Unity client. Finally, the *ActionManager* receives this command and triggers the corresponding *ICharacterBehavior* on Wurguth's *VivCharacter* com-

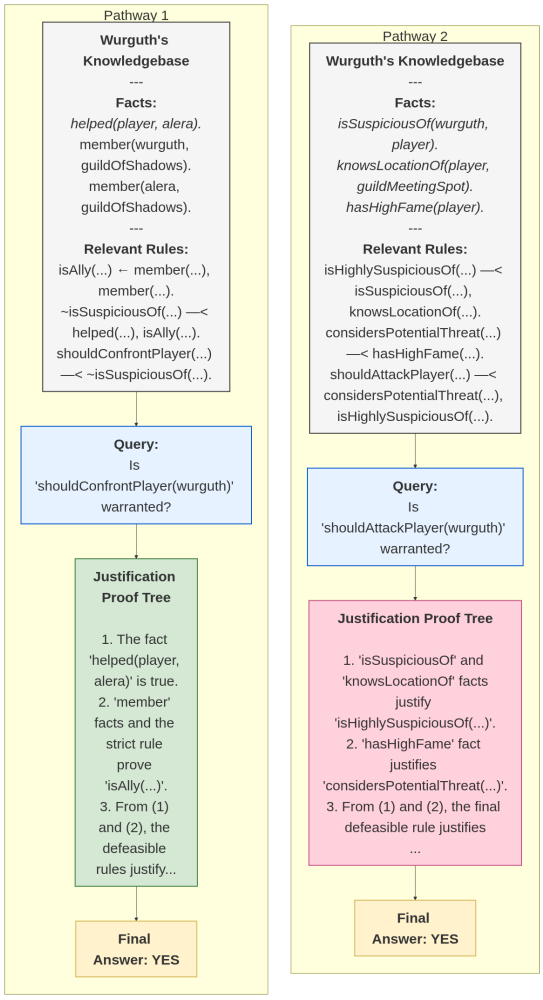

Figure 5: A comparison of the DeLP reasoning process showing how different runtime facts lead to distinct outcomes. Left (Pathway 1): A positive social fact added to the knowledge base warrants a non-hostile 'confront' behavior. Right (Pathway 2): A series of suspicious facts added to the knowledge base warrants a hostile 'attack' behavior.

ponent, causing him to physically approach the player in the game world to initiate a dialogue and complete the full reasoning-to-action loop.

Pathway 2: Snooping As Usual...

In contrast, the second pathway illustrates how the system generates a much more hostile outcome when the player's actions suggest a direct threat to the guild. In this scenario, the player ignores Alera's quest at first and instead interrogates locals about the Guild's activities. Successful skill checks in conversations with two separate individuals result in the player learning sensitive information, triggering specific *KnowledgeUpdateRule* assets that add the corresponding facts to Wurguth's knowledge base. Wurguth, monitoring these activities, has his *DELPEntity* updated after each event. First, a fact is added indicating he is suspicious of the player. This is escalated when he learns the player knows the location of the meeting spot, adding a fact representing high suspicion. Finally, a separate, highly visible combat encounter adds a fact about the player's high fame, which a background rule uses to derive that Wurguth now considers the player a potential threat.

With this new set of facts, Wurguth's reasoning process during the *Supertask* re-evaluation yields a different result. As shown in our diagram (Fig. 5, Pathway 2), the arguments for a non-hostile response are no longer valid. Instead, the combination of Wurguth's high suspicion and his consideration of the player as a potential threat provides the justification needed to warrant the conclusion that he should attack.

The strategic layer's selection algorithm now finds both *investigateSuspiciousActivity* and *neutralizeThreat* to be valid *Behaviors*. As per Wurguth's *VivPersona*, *investigateSuspiciousActivity* has the higher priority and is added to the dispatch list first. Because *neutralizeThreat* is defined as compatible with this behavior, it is also added to the list. The *Viv* manager consequently dispatches a *VivWME* containing both behaviors to the ABL server. The ABL agent begins executing these tactics in parallel, including the *neutralizeThreat* tactic which uses a primitive act for an AggressiveConfrontation. This command is sent back to Unity, triggering the corresponding *ICharacterBehavior* on Wurguth's *VivCharacter* component. This final step results in Wurguth engaging the player with hostile actions and dialogue, and will trigger a combat encounter unless the player can pass a difficult skill check to de-escalate the situation—a stark contrast to the outcome of the first pathway, driven entirely by the different facts accrued from player actions.

Discussion

The architecture presented in this paper achieves a novel synthesis of performance theory, formal logic, and dynamic behavior execution. As the case study illustrates, this integration enables the creation of nuanced, context-sensitive character behaviors that balance both coherence and flexibility. In this section, we therefore analyze the primary benefits and limitations of this approach and discuss its broader implications for the design of believable virtual agents.

Architectural Benefits

The architecture presented in this paper was designed to address the three core research questions posed in our introduction. The system's design and the outcomes of the case study provide insight into each of these questions.

First, we asked what formalisms can be drawn upon to model the nuances of believable, character-driven performance. We argue that a powerful answer lies in a synthesis of formalisms from both performance theory and symbolic AI. Our architecture operationalizes this by computationally modeling a character's "internal monologue," a process akin to the Stanislavskian actor's method of internal justification.

Second, we asked how an agent's actions can remain coherent with its high-level goals while retaining the flexibility to react to novel situations. The "internal monologue" model addresses this inherent tension. In our system, the DELP

reasoning layer provides the foundation for the monologue, allowing the agent to flexibly form competing arguments based on its dynamic knowledge. The ABL tactical layer offers a rich vocabulary of behaviors with which to express the monologue's outcome. Finally, the strategic *Viv* component acts as the arbiter, ensuring coherence by selecting a behavioral plan from the valid options that is most consistent with the character's *VivPersona*.

Finally, our third research question addressed the dual challenges of achieving deliberative transparency while mitigating the prohibitive authorial costs associated with highly specified characters. Our architecture provides transparency at two distinct levels. At the reasoning layer, the dialectical process of DeLP yields a formal, traceable proof for every decision, as illustrated in our case study (Figure 3). This is complemented at the tactical level by the ABL debugger, which provides a real-time view of the agent's active goal tree. The mitigation of authorial burden is addressed through a deliberate commitment to a modular, asset-based design. Key components of a character's identity—its knowledge (*DELPEntity*) and personality (*VivPersona*)—are implemented as ScriptableObject assets, allowing designers to create and configure new character archetypes directly within the Unity editor. Furthermore, the "hivemind" approach of the ABL tactical layer ensures that once a behavior like investigateSuspiciousActivity is implemented, it is reusable by any Viv-controlled character and ensures that the high initial authoring cost attenuates over time.

Limitations and Engineering Challenges

Despite its benefits, the architecture in its current form has some limitations. While we strove primarily to reduce the amount of authoring needed, the complexity of authorship leaves something to be desired. Adding a new primitive action is a meticulous, multi-step process across the client-server divide. A similar challenge exists in knowledge engineering. Authoring effective *KnowledgeUpdateRule*s and defeasible programs requires advanced familiarity with logic. Furthermore, the limitless expressive power of logic necessitates a highly disciplined and planned ontology for the game's domain to prevent the knowledge base from becoming unmaintainable.

From a performance perspective, the deliberative loop through the external server is not designed for split-second reflexes. While the underlying ABL tactical layer is capable of such actions, the current architecture lacks a "reflex arc" to bypass the strategic layer, which remains an avenue for future work. Finally, system performance at scale is an untested, open question. The computational cost of the DeLP solver and the network traffic generated by a large number of concurrent agents are likely to present challenges.

Broader Implications and Future Work

We argue, ultimately, for a shift in design philosophy, moving from the direct scripting of agent actions toward the authoring of an agent's underlying "mind." By focusing on the interplay between a character's knowledge, reasoning process, and personality, designers are empowered to create agents capable of a wider range of emergent, yet coherent, behaviors. The use of ScriptableObject assets for components like the *DELPEntity* and *VivPersona* represents a practical step toward making this authoring paradigm more accessible to designers.

Given the system in its current state, we have identified several thrusts for future work aimed at addressing its limitations and extending its capabilities. The first is to mitigate the authoring complexity through the development of a suite of dedicated editor tools designed to streamline the creation of new *Behaviors* and provide a more intuitive interface for authoring defeasible logic rules. A second, more ambitious thrust involves extending the *VivPersona* to incorporate dynamic character traits. This is not an end in itself, but a step toward a much larger architectural goal: integrating dynamic goal-setting. A more robust persona model could provide the basis for a system that autonomously selects its own *Supertask* based on its evolving character and the narrative context. We recognize that synthesizing goal-selection with goal-execution in this way represents a considerable research effort, but it would be a large leap forward for the system's autonomy.

Finally, beyond extending the system itself, a rigorous empirical validation is necessary. A formal comparative evaluation should be conducted not only against traditional static Behavior Trees but also against other deliberative paradigms like GOAP. Furthermore, evaluating this symbolic, "glass box" approach against more modern, opaque systems—such as those driven by large language models or other generative machine learning techniques—would be a valuable contribution to understanding the trade-offs between competing AI design philosophies.

Conclusion

Creating believable artificial agents with coherent inner lives remains one of the most compelling challenges in artificial intelligence. The architecture detailed herein is a novel approach, synthesizing principles from Stanislavski's performance theory, the formal argumentation of defeasible logic, and the dynamic execution of specialized agent languages. The primary contribution of this work is the grounding of an expressive execution engine (ABL) in a formal logic of internal conflict (DeLP). This provides a computational analogue for a character's internal monologue, allowing for choices that are not merely computed, but justified from the agent's own perspective. We believe this architecture charts a compelling new path toward agents that can finally begin to bridge the gap between simulation and soul.

Acknowledgements

We would like to acknowledge the support of the National Science Foundation under Grant No. IIS-2232066, as well as UC Davis and the Center for Artificial Intelligence and Computational Futures.

References

Bratman, M. E. 1987. *Intention, Plans, and Practical Reason*. Cambridge, MA: Harvard University Press.

Cavazza, M.; Lugrin, J.-L.; Pizzi, D.; and Charles, F. 2007. Madame Bovary on the holodeck: immersive interactive storytelling. In *Proceedings of the 15th ACM International Conference on Multimedia*, 651–660. Augsburg, Germany: ACM. ISBN 978-1-59593-701-8.

Colledanchise, M.; and Ögren, P. 2018. *Behavior Trees in Robotics and AI: An Introduction*. Chapman 'I&' Hall/CRC Artificial Intelligence and Robotics Series. Boca Raton, FL: CRC Press, 1st edition. ISBN 9781138593732.

Dias, J.; Mascarenhas, S.; and Paiva, A. 2014. FAtiMA Modular: Towards an Agent Architecture with a Generic Appraisal Framework. In *Proceedings of the 7th International Workshop on Emotional and Empathic Agents*, AAMAS '14.

El-Nasr, M. S.; Yen, J.; and Ioerger, T. R. 2000. FLAME—Fuzzy Logic Adaptive Model of Emotions. *Autonomous Agents and Multi-Agent Systems*, 3(3): 219–257.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN Planning: Complexity and Expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, 1123–1128. AAAI Press.

Evans, R.; and Short, E. 2014. Versu: A Simulationist Storytelling System. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2): 113–130.

García, A. J.; and Simari, G. R. 2004. Defeasible logic programming: An argumentative approach. *Theory and practice of logic programming*, 4(1-2): 95–138.

Georgeff, M. P.; and Lansky, A. L. 1987. Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, 687–691. Seattle, WA: AAAI Press.

Guzdial, M.; and Riedl, M. 2018. Automated Game Design via Conceptual Expansion. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 14, 31–37.

Horswill, I. 2022. Step: A Highly Expressive Text Generation Language. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Pomona, CA.

Horswill, I.; and Hill, S. 2024. Fast, Declarative, Character Simulation Using Bottom-Up Logic Programming. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 20, 54–64.

Junius, N.; Mateas, M.; Wardrip-Fruin, N.; and Carstensdottir, E. 2022. Playing with the Strings: Designing Puppitor as an Acting Interface for Digital Games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, 250–257.

Laird, J. E.; Newell, A.; and Rosenbloom, P. S. 1987. SOAR: An Architecture for General Intelligence. *Artificial Intelligence*, 33(1): 1–64.

Mateas, M. 2001. Expressive AI: A Hybrid Art and Science Practice. *Leonardo*, 34(2): 147–153.

Mateas, M.; and Stern, A. 2002. A Behavior Language for Story-Based Believable Agents. *IEEE Intell. Syst.*, 17: 39–47.

Mateas, M.; and Stern, A. 2003. Façade: An Experiment in Building a Fully-Realized Interactive Drama. In *Game Developers Conference*. Game Design Track.

Mateas, M.; and Stern, A. 2004. A Behavior Language: Joint Action and Behavioral Idioms. In Prendinger, H.; and Ishizuka, M., eds., *Life-Like Characters: Tools, Affective Functions, and Applications*, 135–161. Berlin, Heidelberg: Springer.

McCoy, J.; Treanor, M.; Samuel, B.; and Wardrip-Fruin, N. 2011. Comme il Faut: A System for Authoring Playable Social Models. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2011, October 10-14, 2011, Stanford, California, USA*, 158–163. AAAI Press.

Meehan, J. R. 1977. TALE-SPIN, An Interactive Program that Writes Stories. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, 91–98.

Mitchell, K. D.; and McCoy, J. 2024. Exploring Stanislavskian Performance for Agent-based Nonplayer Characters through Defeasible Logic. In *Proceedings of the 24th ACM International Conference on Intelligent Virtual Agents*, IVA '24. New York, NY, USA: Association for Computing Machinery. ISBN 9798400706257.

Orkin, J. 2006. Three States and a Plan: The A.I. of F.E.A.R. In *Game Developers Conference*.

Schank, R. C.; and Abelson, R. P. 1977. *Scripts, Plans, Goals, and Understanding: An Inquiry Into Human Knowledge Structures*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Sekhavat, Y. A. 2017. Behavior Trees for Computer Games. *International Journal of Artificial Intelligence Tools*, 26(02).

Si, M.; Marsella, S. C.; and Pynadath, D. V. 2005. THESPIAN: An Architecture for Interactive Pedagogical Drama. In *Proceedings of the 12th International Conference on Artificial Intelligence in Education (AIED2005)*, 616–623.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. The MIT Press, 2nd edition.

Thimm, M. 2017. The Tweety Library Collection for Logical Aspects of Artificial Intelligence and Knowledge Representation. *Künstliche Intelligenz*, 31(1): 93–97.

Wang, L.; Ma, C.; Feng, X.; Zhang, Z.; Yang, H.; Zhang, J.; Chen, Z.; Tang, J.; Chen, X.; Lin, Y.; Zhao, W. X.; Wei, Z.; and Wen, J.-R. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6): 186345.

Ware, S. G.; and Siler, S. 2021. Sabre: A Narrative Planner Supporting Intention and Deep Theory of Mind. In *Proceedings of the Seventeenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 17, 99–106. AAAI Press.

Ware, S. G.; and Young, R. M. 2014. Glaive: A State-Space Narrative Planner Supporting Intentionality and Conflict. In *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 80–86. AAAI Press.