



**Università
di Genova**

DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Regia: A Domain Specific Language to expand usability of VEsNA-ProC

Vassallo Edoardo

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy

<https://www.dibris.unige.it/>



MSc Computer Science
Data Science and Engineering Curriculum

Regia: A Domain Specific Language to expand usability of VEsNA-ProC

Vassallo Edoardo

Advisor: Prof. Viviana Mascardi
Dott. Fabio Casale

Examiner: Prof. Giovanni Lagorio

September, 2026

*A mia sorella, ai miei genitori, ai miei familiari, ai miei amici.
A tutti coloro che mi supportano e sopportano ogni giorno.*

Abstract

Non-playable characters in narrative games are usually specified by designers, in prose, and implemented by programmers, in code the designers cannot verify. The Belief-Desire-Intention model describes agents through the notions designers already use for characters, their knowledge, goals, and commitments, but its programming languages assume a programming background, while the accessible authoring tools common in game development offer only reactive behaviour.

This thesis introduces Regia, a domain-specific language through which designers write deliberative agents as part of a story. Developed together with the studio Untold Games, the language takes the scenario, rather than the single agent, as its unit of description. Plots divide a story into phases; Roles stand for the parts agents play in it; Playbooks package reactive behaviour that a Plot assigns to a Role and later withdraws. Keywords replace symbolic operators, and programs contain no variables. Regia programs are transpiled into AgentSpeak and executed by the Jason interpreter inside the Godot engine, through the VEsNA framework, while a web-based editor displays the phases and transitions of each story as an editable graph.

The language was assessed through a game developed entirely in Regia, synthetic benchmarks, a comparison with agent-oriented methodologies based on a shared metamodel, and a questionnaire with 26 participants. Compilation time scales close to proportionally with the length of the Regia source, and is not driven by the volume of generated code. Regia covers the runtime concepts of agent systems more fully than their design-time concepts, in line with its focus on narrative. Participants unfamiliar with the language, most with a computing background, read it accurately, and their errors in writing it were mostly slips of syntax or declaration; the Role construct was the main source of misunderstanding. An evaluation with designers who do not program is left to future work.

Table of Contents

Declaration on the Usage of AI Tools	13
Chapter 1 Introduction and Motivation	14
Chapter 2 Background	17
2.1 Game AI: Standard Behaviour Models	17
2.1.1 Finite State Machines	17
2.1.2 Behaviour Trees	18
2.1.3 Goal-Oriented Action Planning	19
2.1.4 Hierarchical Task Networks	20
2.1.5 Reinforcement Learning	20
2.1.6 Shared Assumptions and the Authoring Gap	20
2.2 Agent-Oriented Programming and the Belief-Desire-Intention Model	21
2.2.1 Practical Reasoning and Intentions	21
2.2.2 The BDI Architecture	21
2.2.3 Agent-Oriented Programming	23
2.3 AgentSpeak and Jason	23
2.3.1 Beliefs	23
2.3.2 Goals and Events	24
2.3.3 Plans and Execution	25
2.3.4 Plan Selection and Intentions	25

2.3.5	Jason as a Concrete Interpreter	26
2.3.6	Plan Labels and Annotations	26
2.3.7	The Environment Interface	27
2.3.8	Internal Actions	27
2.4	Interactive Narrative and Drama Management	27
2.4.1	The Agency-Coherence Tension	28
2.4.2	Two-Tier Architectures and the Authoring Bottleneck	28
2.5	Domain-Specific Languages	29
2.6	Integration Architecture: VEsNA and Godot	30
2.6.1	The Real-Time Engine Environment and the Impedance Mismatch	30
2.6.2	VEsNA: Goals and Architecture	30
Chapter 3	Related Work	32
3.1	Model-Driven Game Development and Authoring Tools	32
3.1.1	Visual Scripting and Reactive Logic	33
3.1.2	Narrative Domain-Specific Languages	33
3.1.3	Interactive Drama Authoring: Scribe	34
3.2	Methodologies and Modelling Languages for BDI Development	34
3.3	Agent-Engine Integration Middleware	35
3.4	Summary	36
Chapter 4	Requirements and Analysis	38
4.1	Stakeholders and Methodology	38
4.2	Syntax and Accessibility	39
4.2.1	Notation	39
4.2.2	Vocabulary	40
4.3	From Agents to Narratives	41
4.3.1	Snippets	41

4.3.2	Making the Narrative Central	43
4.4	Dynamic Behaviour	43
4.5	Visualisation	44
4.6	Modularity and Reuse	45
4.6.1	Practical Hygiene	45
4.6.2	Composability	45
4.7	Summary	46
Chapter 5	Language Design	47
5.1	Program Definition	50
5.1.1	Comments	51
5.2	Vocabulary: Facts, Actions, Events	51
5.2.1	Facts	52
5.2.2	Actions	52
5.2.3	Events	53
5.3	Playbooks	54
5.3.1	Plans	55
5.3.2	Statements	55
5.3.3	Conditional Branches	57
5.3.4	Conditions	58
5.3.5	Priority and Temper	58
5.4	Plots	59
5.4.1	Phases and Roles	60
5.4.2	DURING Blocks	61
5.4.3	Transitions	62
5.4.4	Plot Termination	63
5.4.5	Director WHEN Blocks	63
5.4.6	Director Statements	64

5.4.7	Subplots	66
Chapter 6	Implementation	68
6.1	Transpiler	68
6.1.1	Architecture and Pipeline	68
6.1.2	From Source to AST	70
6.1.3	Semantic Validation	72
6.1.4	AgentSpeak Emission	72
6.1.5	Diagnostics	76
6.1.6	Invoking the Compiler	78
6.1.7	Testing	78
6.2	Visual Editor	78
6.2.1	Overview and Use	79
6.2.2	Source to Graph	80
6.2.3	Graph to Source	82
6.2.4	Error Handling and Limitations	82
Chapter 7	Experiments	84
7.1	“Everything For Science!” Test Case	84
7.1.1	Infrastructure	84
7.1.2	Game Plot	88
7.1.3	Results	91
7.2	Transpiler Benchmarks	93
7.2.1	The Benchmark Tool	93
7.2.2	Scaling Results	96
7.2.3	Scaling Interactions	98
Chapter 8	Language Evaluation	101
8.1	Evaluation Approach	101

8.2	Comparison with the FAML Reference Metamodel	102
8.2.1	Method	102
8.2.2	Results	103
8.2.3	Interpretation	103
8.3	Questionnaire	104
8.3.1	Instrument	104
8.3.2	Participants and Procedure	106
8.3.3	Metrics	106
8.3.4	Results	109
8.4	Threats to Validity	113
8.4.1	Metamodel Comparison	113
8.4.2	Questionnaire	114
Chapter 9	Discussion and Conclusions	115
9.1	Discussion	115
9.1.1	Requirements Revisited	115
9.1.2	Accessibility	115
9.1.3	Deliberation and Expressiveness	117
9.1.4	Embodiment and Practicality	118
9.1.5	Regia Against Related Work	119
9.1.6	Applicability Beyond Games	119
9.2	Limitations and Future Work	119
9.3	Conclusions	120
Appendix A	FAML Metamodel Mapping	122
Appendix B	Default Benchmark Program	130
Bibliography		133

Declaration on the Usage of AI Tools

In accordance with the University's guidelines on the use of artificial intelligence, I declare the following uses of generative AI tools in this work.

Software. During the development of the transpiler and the Visual Editor, I used GitHub Copilot, an AI coding assistant integrated in Visual Studio Code, for code completion, updating automated test cases, and for refactoring code I had written.

Text. I used Claude (Opus 5.5), a large language model developed by Anthropic, to review the manuscript against the style guidelines of the department, to identify inconsistencies and errors, to shorten and restructure passages, and to propose rewordings.

Review and responsibility. All output of these tools, code and text alike, was reviewed, verified, and where necessary corrected by me, and I take full responsibility for the content of this thesis. The research questions, the design of the language, the collection of the data, and the interpretation of the results are my own. No personal data nor confidential documents were provided to these tools.

Chapter 1

Introduction and Motivation

“Is it really true?
That a bird like me could never think like you?”

— *from “BIRDBRAIN.”*
Jamie Paige Featuring OK Glass and 重音テト.
Constant Companions. Sunrise 101.1, 2025.

Stories have always been an essential part of human life. Myths helped explain the world around us. Fables taught the young and transmitted values. Novels entertained our ancestors as they still entertain us today.

Ever since we began telling them, stories have been limited by two things: the imagination of their creator, and the quirks of the medium containing them. Novels and other written forms rely on the mastery of words, sentences, and descriptions. Movies and plays require a director to communicate a vision to actors, and those actors to portray the emotions of their characters convincingly. The videogame, a newer medium, sets further limits on authors through its complex and multi-faceted nature.

The first of these limits is interactivity. Unlike other media, games expect their consumers, the players, to interact with the game world and to change its narrative, through decisions ranging from moment-to-moment choices to those that alter the outcome of the story entirely. The simple requirement that a game could be won or lost already creates a first branch in that narration. Authors must therefore decide how much autonomy to grant the player. A linear game, in which the player follows a predetermined narrative, is easier to construct, but it reduces the impact of the player’s actions on the world they inhabit. A narrative that accounts for player input is far more complex to author, and much of what it contains may never be seen by any single player. Balancing the two calls for systems

that carry part of that burden, by generating portions of the narrative at runtime.

The second limit lies in how a game is made. A writer or designer may conceive of their game as a story, with characters and plot beats, but the product is ultimately software. In smaller development teams, writers and programmers are often the same people. In larger studios the two roles are held by different ones, with different backgrounds and different perspectives on what the game represents. A designer is responsible for defining the game, its mechanics, and its narrative, yet each decision must pass through the programmer who implements it [39]. The result is friction between the two. Designers may pose requests that are difficult to implement, while a programmer’s design choices may limit the number of features a designer can express. The two rarely share a formal notation: designers typically describe their narratives in natural language, as prose, which leaves room for misinterpretation between two parties without a common vocabulary.

One element that often suffers in this translation is the Non-Playable Character (NPC), the computer-controlled actor that participates in the story alongside the player. Many games present simple narratives, with few characters or none at all. Other genres, Role-Playing Games among them, build far more extensive stories around such characters, whether as enemies or as allies. The split between the world views of designers and programmers shows in how each side characterises them. Designers see NPCs in humanistic terms, and describe their behaviour through traits: what a character wants, and how it reasons about obtaining it. Programmers see the system they will implement, a reactive agent that responds to immediate events, without goals or memories.

These concerns were already a topic of discussion among our collaborators at Untold Games [57], a Genoa-based game studio. They had already experimented with written and graphical systems for formalising their designers’ narratives. During a collaboration with our department, as part of the VEsNA project [19], the studio encountered the Belief-Desire-Intention model (BDI) and its anthropomorphic approach to modelling agents. That perspective sat closer to the designers’ own conceptualisation of a character, and came with implementations capable of authoring multi-agent systems directly.

Obstacles remained. Few integrations exist between game engines and BDI systems, a problem the VEsNA project addresses. The more pressing obstacle is accessibility. The languages implementing BDI, AgentSpeak among them, are declarative and logic-based, and their syntax and inner workings present a barrier to programmers unfamiliar with the paradigm, and a greater one to game designers. A simpler language was therefore needed, and building one is the goal of this thesis.

We present Regia, a domain-specific language for modelling agent behaviour, addressed to game designers. Regia provides a bridge representation between designers and programmers. Its syntax and structure sit at a higher level than AgentSpeak’s, and stay readable by someone who does not program, while remaining formal enough to translate directly

into AgentSpeak under a single clear interpretation. We realise the language through a transpiler and a visual editor. Both, together with the benchmarks and the case study reported in Chapter 7, are publicly available [59]. The remainder of this thesis follows the development process, from the problem that motivates it to the evaluation of what was built.

Chapter 2 covers the two fields this work joins: how autonomous behaviour is built in games today, and the Belief-Desire-Intention model together with the languages and platforms that bring such agents into a game engine.

Chapter 3 surveys existing systems for authoring agent and narrative behaviour, considering accessibility and game engine integration.

Chapter 4 derives the language requirements detected during our collaboration with Untold Games, and states what falls outside its scope.

Chapter 5 presents the language construct by construct, giving the grammar of each alongside the AgentSpeak it compiles to, carried throughout by one running example.

Chapter 6 describes the two artifacts that realize it, the transpiler and the visual editor.

Chapter 7 builds a complete game with Regia, and measures how the transpiler behaves as a project grows.

Chapter 8 evaluates the language against an established evaluation framework for agent-oriented languages, adapted to our needs.

Chapter 9 discusses these results together with those of the experiments, returns to the requirements of Chapter 4, and closes with the limitations left open, the work they suggest, and the conclusions of the thesis.

Chapter 2

Background

In this chapter we introduce the concepts this thesis builds on. Section 2.1 presents the methods the videogame industry uses to build intelligent agents. Section 2.2 contrasts them with the Belief-Desire-Intention model, and Section 2.3 presents its implementation through AgentSpeak and Jason. Section 2.4 discusses drama management in the game design literature, Section 2.5 domain-specific languages, and Section 2.6 the VEsNA project.

2.1 Game AI: Standard Behaviour Models

Game designers implement agent behaviour with an established set of methods, from finite state machines to planning-based techniques. This section introduces each paradigm, its authoring model, and its known limitations, which Section 2.2 contrasts with the deliberative reasoning of the BDI model.

2.1.1 Finite State Machines

A finite state machine (FSM) represents an agent's behaviour as a set of discrete states, together with transitions between them [43]. For example, an agent implementing a guard may possess states such as `Patrol`, `Chase`, or `Attack`. At any point in time exactly one state is active, and the agent executes only the logic associated with it. Transitions fire when a specified condition or event occurs, moving the agent from one state to another. Figure 2.1 illustrates a simple FSM graph.

FSMs offer two properties that make them attractive to designers. First, their behaviour is easy to predict, since an agent in a given state can only perform the actions associated

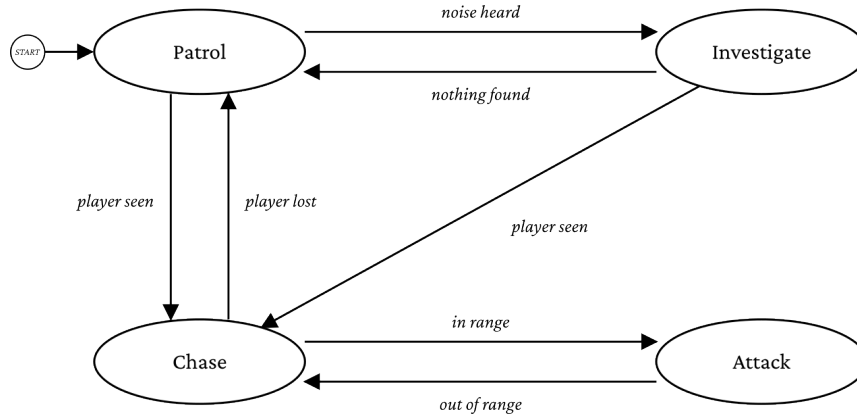


Figure 2.1: A simple finite state machine diagram demonstrating discrete states and explicit event-driven transitions.

with it. Second, the model maps onto a visual notation that most designers can read and edit without programming experience.

The model does not scale well: as states are added, the number of transitions grows quadratically, since every state may need a transition to every other relevant state [43]. This effect, commonly called *state explosion*, forces designers to revisit existing transitions whenever they add a state, risking regressions in behaviour that previously worked. A large FSM also becomes hard to visualise, since its states and transitions are too many to inspect as a whole.

2.1.2 Behaviour Trees

Behaviour trees address the maintainability problems of finite state machines by organising behaviour hierarchically rather than as a flat graph of states [26]. A behaviour tree consists of composite nodes, decorator nodes, and leaf nodes, as shown in Figure 2.2. Composite nodes control how their children execute: a sequence node runs its children in order and fails as soon as one child fails, while a selector node tries its children in order and succeeds as soon as one child succeeds. Decorator nodes wrap a single child and modify its behaviour or return value, for example by inverting a success into a failure or by imposing a cooldown between executions. Leaf nodes represent atomic actions or condition checks and return one of three statuses: success, failure, or running. The tree evaluates once per simulation step, starting from the root and propagating status values back up through the hierarchy [26], a process game developers call a *tick*.

Behaviour trees sit at a lower level of abstraction than FSMs, and may appear less accessible. They offer, however, two advantages that matter in larger productions. Because subtrees are self-contained, designers can reuse and recombine them without altering un-

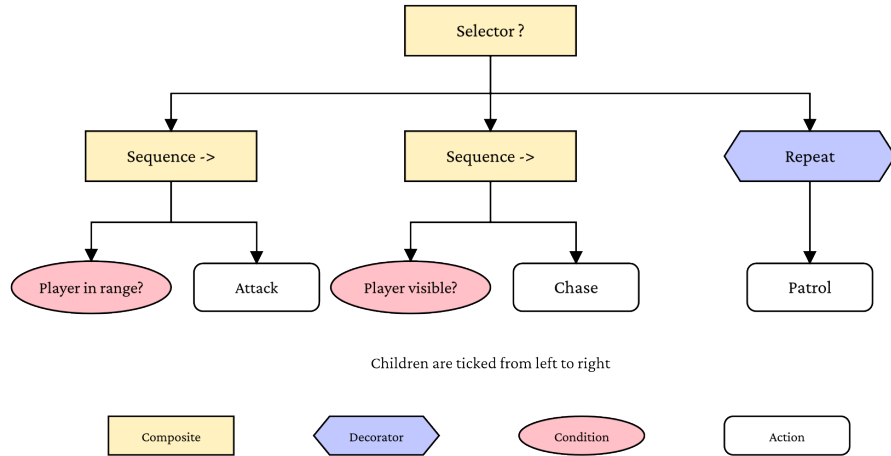


Figure 2.2: A behaviour tree structure detailing the hierarchical arrangement of composite, decorator, and leaf nodes.

related parts of the tree, which mitigates the state explosion of Section 2.1.1. The tree structure is also at least as easy to visualise as a state graph. Since their popularisation in *Halo 2* (2004) [27], behaviour trees have become a standard authoring model for game AI, and mainstream engines such as Unreal Engine provide visual, node-based editors for them [63].

2.1.3 Goal-Oriented Action Planning

Goal-Oriented Action Planning (GOAP), introduced for the game *F.E.A.R.* (2005), shifts the authoring burden from explicit control flow to a declarative description of available actions [44]. Rather than authoring transitions directly, the designer defines a set of atomic actions, annotating each one with preconditions and effects expressed over the current state of the game world, and defines one or more goals describing a desired world state.

At runtime, a planner searches the space of available actions for a sequence that transforms the current world state into the goal state, constructing this plan dynamically rather than following a path authored in advance. Consequently, GOAP-driven agents can produce behaviour that the designer did not explicitly anticipate, while still respecting the constraints encoded in the preconditions and effects of each action [44].

GOAP thus moves the designer’s effort from specifying control flow to specifying actions and goals. The same flexibility reduces authorial control: a designer cannot easily steer an agent towards a preferred behaviour, and the planner may produce behaviour that is valid but uninteresting.

2.1.4 Hierarchical Task Networks

Hierarchical Task Network (HTN) planning offers an alternative to the bottom-up search used by GOAP. Instead of searching directly over primitive actions, an HTN planner decomposes a compound task into subtasks through designer-defined decomposition methods [15], recursing until only primitive, directly executable tasks remain.

Behaviour trees and HTNs are both hierarchical, but their execution models differ. A behaviour tree ticks reactively, adapting to the world state at every step. An HTN planner plans ahead [32], producing a complete sequence of primitive actions before execution begins. If the world changes significantly during execution, the HTN must discard its plan and replan. Top-down decomposition resembles how designers describe complex behaviour, from a broad objective to its steps, but trades the reactivity of behaviour trees for deeper planning.

2.1.5 Reinforcement Learning

Reinforcement Learning (RL) departs from the explicit authoring models above. Rather than authoring states, trees, or actions, the designer defines a reward function, and the agent learns a policy through trial and error that maximises its cumulative reward [63]. Engines increasingly support this paradigm, for example through Unity ML-Agents [29]. RL conflicts, however, with the authorial control game design requires [63]. The learned behaviour is typically encoded in the weights of a neural network, a black box: if an agent acts undesirably during a narrative beat, the designer cannot inspect or adjust its logic, only change the reward function or the training environment and retrain, with no guarantee that the fix preserves other learned behaviour. RL excels in continuous control and locomotion, but is poorly suited to orchestrating precise, narrative-driven character interactions.

2.1.6 Shared Assumptions and the Authoring Gap

The paradigms of this section show a trade-off between authorability and deliberative depth. Reactive models such as behaviour trees give designers direct control but do not plan ahead. Planning models such as GOAP and HTN look ahead, but pursue one goal or task at a time, chosen by designer-defined relevance rather than weighed against competing motivations. Reinforcement Learning adapts to its environment, but hides its logic from the designer.

The paradigms also share an assumption: they model an agent as a controller that selects actions, not as a character with motives. In games with a narrative, many agents are

characters, and, as Chapter 1 noted, designers describe them in terms of what they know, what they want, and what they have decided to do. None of the paradigms above represents these notions explicitly. Section 2.2 introduces the Belief-Desire-Intention model, which represents them as distinct mental attitudes, while keeping the agent’s logic explicit.

2.2 Agent-Oriented Programming and the Belief-Desire-Intention Model

The Belief-Desire-Intention (BDI) model represents an agent’s information, motivations, and commitments as distinct mental attitudes, while keeping the agent’s logic explicit and authorable. This section introduces the philosophical origin of the model, its three mental attitudes, the reasoning cycle that connects them, and the wider paradigm of agent-oriented programming it belongs to.

2.2.1 Practical Reasoning and Intentions

The BDI model originates in Michael Bratman’s account of practical reasoning, reasoning directed at deciding what to do rather than what to believe [9]. Bratman argues that intentions are a distinct mental attitude that cannot be reduced to desires. A desire expresses something an agent would like to achieve, without commitment; an intention expresses a desire the agent has committed to pursuing. Once adopted, an intention constrains further deliberation: the agent does not normally reconsider it unless new information calls it into question.

This distinction between desiring and committing matches how designers describe characters (Section 2.1.6), and is absent from the paradigms of Section 2.1. A behaviour tree, for example, re-evaluates its hierarchy at every tick and reacts only to the current state. It has no explicit representation of what the agent is committed to, beyond whichever leaf last returned a running status.

2.2.2 The BDI Architecture

Georgeff, Pell, Pollack, Tambe, and Wooldridge formalised Bratman’s distinctions into a computational architecture built around three data structures [22].

2.2.2.0.1 Beliefs. Beliefs represent the agent’s information about itself and its environment. A belief does not need to be objectively true: it represents what the agent currently

takes to be the case, and the agent revises it as new perceptions arrive.

2.2.2.0.2 Desires. Desires represent the states of the world the agent would like to bring about. An agent may hold several desires at once, and they may conflict, since nothing in the model requires them to be jointly achievable.

2.2.2.0.3 Intentions. Intentions represent the desires the agent has selected to pursue. Adopting intentions narrows the agent’s desires down to those it is committed to acting upon. This provides the stability practical reasoning requires: an agent that reconsidered every intention at every step would spend its effort on deliberation rather than action.

A control loop, the BDI reasoning cycle, connects the three structures. At each iteration, the agent perceives its environment and updates its beliefs. It then generates new desires, or options, consistent with its beliefs. Through deliberation, it selects among competing desires and commits to some of them as intentions. Through means-end reasoning, it selects and executes actions that work towards its intentions, and the cycle repeats. Figure 2.3 illustrates the cycle.

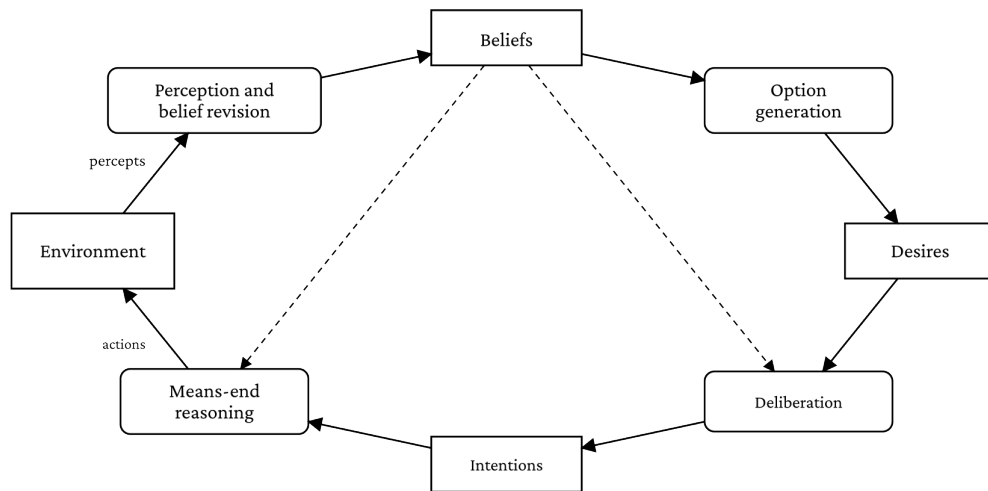


Figure 2.3: The Belief-Desire-Intention reasoning cycle. Perception updates beliefs, deliberation selects intentions from competing desires, and means-end reasoning selects actions consistent with current intentions.

Intentions persist across cycles until they are fulfilled, become impossible, or are dropped. This persistence lets a BDI agent pursue a long-term commitment while still reacting to new events: a new belief can generate a new desire, or revise an existing intention, without discarding the intentions already in progress.

2.2.3 Agent-Oriented Programming

The BDI architecture belongs to a broader paradigm, agent-oriented programming, in which developers decompose software into autonomous agents rather than objects or procedures [28]. An agent is a computational entity that perceives its environment, maintains an internal state, and decides autonomously which actions to take towards its objectives, running continuously without an external controller invoking it at each step. Jennings identifies autonomy, social interaction, and reactivity to a changing environment as the properties that distinguish agent-oriented software from conventional software components [28].

Several computational realisations of the BDI architecture exist, differing mainly in how they represent and manipulate beliefs, desires, and intentions. Section 2.3 presents AgentSpeak, the BDI language this thesis targets and the compilation target of the transpiler described in Chapter 6.

2.3 AgentSpeak and Jason

Section 2.2 described the BDI model as a conceptual architecture, without a concrete syntax. AgentSpeak provides this syntax [52]. We describe it as implemented by Jason, the interpreter developed by Bordini, Hübner, and Wooldridge [8], rather than as Rao’s original specification. Jason is the compilation target of the transpiler of Chapter 6, and provides several extensions this thesis relies on.

Jason is also one of the three technologies of JaCaMo, a platform for multi-agent programming [7]. JaCaMo combines Jason, which programs the agents, with CArtAgO, which models the shared environment as a set of *artifacts*, objects exposing observable properties that agents perceive and operations that agents invoke, and with Moise, which specifies how agents are organised into groups and missions. A JaCaMo system is configured through a `.jcm` file, which declares its agents, their source files, and the environment they share.

This section presents the core constructs of AgentSpeak in BNF, from beliefs to complete plans, and then the features Jason adds.

2.3.1 Beliefs

An AgentSpeak agent represents its beliefs as first-order logical atoms. The following production rules define the structure of a belief literal and its constituent terms:

```

1 <literal> ::= <atom> | <atom> "(" <term-list> ")"
2 <term> ::= <atom> | <variable> | <number> | <string>

```

A literal consists of a predicate name and an optional list of arguments. Variables begin with an uppercase letter and stand for values the agent determines during reasoning. Atoms begin with a lowercase letter and represent fixed symbols. The literal `position(player, 10, 5)` represents a specific belief about the player's location, while `hostile(Agent)` uses the variable `Agent` to refer to an unspecified entity.

The belief literals an agent holds form its belief base. Unlike the world states of the planning models of Section 2.1.3, a belief base need not model the environment accurately: it reflects only what the agent takes to be true.

2.3.2 Goals and Events

AgentSpeak distinguishes two kinds of goal, both built on belief literals:

```

1 <goal> ::= "!" <literal> | "?" <literal>

```

An achievement goal, prefixed by `!`, asks the agent to bring about a state of affairs. A test goal, prefixed by `?`, asks the agent to check whether a literal follows from its belief base. Agents do not pursue goals directly: posting a goal generates a triggering event, as does modifying the belief base. The language recognises six triggers:

```

1 <trigger> ::= "+" <literal> | "-" <literal> | "+!" <literal> | "-!"
  <literal> | "+?" <literal> | "-?" <literal>

```

The `+` and `-` symbols denote the addition and deletion of a belief or goal. Every event is placed on an event queue, which the agent processes in order.

2.3.3 Plans and Execution

A plan is the unit of reactive behaviour in AgentSpeak. It links a trigger to a body, guarded by a context:

```
1  <plan> ::= <trigger> ":" <context> "<->" <body> "."
2  <context> ::= "true" | <literal> | "not" <context> | <context> "&"
   <context>
3  | <context> "|" <context> | "(" <context> ")"
4  <body> ::= "true" | <formula> | <formula> ";" <body>
```

The context is a logical condition over the agent's belief base, and the interpreter considers a plan only if its context holds when the triggering event occurs. The body is a sequence of actions, goals, and belief updates, separated by semicolons. The following two plans respond to the same event, a thief becoming visible:

```
1  +visible(Thief) : health(low) <-
2      !flee.
3
4  +visible(Thief) : health(high) <-
5      !attack(Thief).
```

Listing 2.1: Two AgentSpeak plans responding to the same triggering event.

The contexts `health(low)` and `health(high)` are mutually exclusive given the agent's beliefs, so only one plan applies at a time. Context-based selection lets one event produce different behaviour depending on the agent's internal state, without a separate state for every combination, as a finite state machine would need (Section 2.1.1).

2.3.4 Plan Selection and Intentions

When an event reaches the front of the queue, the interpreter collects every plan whose trigger matches it, the relevant plans. It then evaluates their contexts against the belief base, leaving the applicable plans. If several remain, a selection function chooses one. The interpreter places the chosen plan on the agent's intention stack, which represents the agent's commitment to executing its body. If the body posts further goals, the plans they trigger are stacked on top of the current intention: the outer plan is suspended, and

resumes when the inner plan completes. This stack-based execution lets an agent pursue nested goals while preserving the commitment that motivated them.

2.3.5 Jason as a Concrete Interpreter

Rao’s specification left several points open, among them the strategy the selection function should use to choose between applicable plans [52]. Jason, a Java-based interpreter, resolves them [8]: its default selection function chooses the first applicable plan, in the order plans appear in the plan library. Jason also extends the language with strong negation, which lets an agent represent that a literal is false rather than unknown; plan-failure handling, which lets an agent react when an adopted plan cannot complete; and communication between agents through speech acts. A further extension, plan labels and annotations, is described next.

2.3.6 Plan Labels and Annotations

The plan grammar of Section 2.3.3 attaches no metadata to a plan beyond its trigger, context, and body. Jason extends it with an optional label in front of the trigger, carrying an optional list of annotations [8]:

```

1  <plan> ::= ("@" <label> ("[" <annotation> ("," <annotation>)* "]"
    )?)? <plan-body>
2  <annotation> ::= <atom> | <atom> "(" <term> ")"

```

Jason generates a label when none is given; an author supplies one mainly to attach annotations. Annotations are a general, extensible mechanism. Their main purpose is to let a programmer write a custom selection function that consults them when choosing among applicable plans, and an author may define application-specific annotations for this. Jason also reserves four annotations with an interpreter-level meaning [8], two of which matter to this thesis. **atomic** guarantees that once a plan carrying it starts, no other intention is interleaved until it finishes; Chapter 6 uses it to make phase transitions uninterruptible. **priority** is meant to let the default selection function order applicable plans by an integer, higher values first. Bordini, Hübner, and Wooldridge, however, describe it as “as yet unavailable, but a reserved plan annotation” [8]: it belongs to the vocabulary of the specification, but the interpreter ignores it. Sections 5.3.5 and 4.6 return to this limitation.

2.3.7 The Environment Interface

AgentSpeak defines how an agent reasons over its beliefs and executes actions, but not where beliefs come from or what effect an action produces. Jason answers this through the environment interface, a component outside the reasoning cycle that delivers perceptions to the agent and executes its actions [8].

At the start of each reasoning cycle, Jason queries the environment for the current observable literals and compares them with the beliefs the agent holds. A newly exposed literal is added to the belief base and generates a belief addition event; a literal no longer exposed is removed and generates a belief deletion event. An external system can therefore drive an agent's beliefs by exposing literals, without knowledge of the agent's logic.

Action execution works in the opposite direction. When a plan body executes an action, Jason does not interpret its meaning, but forwards the action name and arguments to the environment interface, which produces the effect in the external system. This division lets the same agent logic run in different environments.

2.3.8 Internal Actions

Agents also need computations that are not actions in the environment, such as arithmetic or list manipulation. Jason provides internal actions for these [8]. An internal action is written with a namespaced identifier, such as `math.sqrt(X, R)`, or, for Jason's standard library, with a leading period, such as `.print`. Jason executes an internal action itself, from its own libraries, instead of forwarding it to the environment. Developers can also register custom internal actions, implemented in Java and exposed under a chosen namespace, to extend an agent's capabilities without a round trip through the environment.

2.4 Interactive Narrative and Drama Management

Sections 2.2 and 2.3 described how a single BDI agent perceives, deliberates, and acts. Populating a game with several such agents, however, does not produce a coherent narrative, since nothing in these mechanisms coordinates agents towards a shared story. This section introduces interactive narrative research, which studies this coordination problem, and the drama management architectures it proposes.

2.4.1 The Agency-Coherence Tension

Interactive narrative research studies how a system can present a story while giving the player some agency over how events unfold [53]. This introduces a tension absent from traditional storytelling. A linear story guarantees coherence and pacing, because its author controls every event in advance. An interactive story cannot, since the player’s choices determine which events occur, and a system that reacts to each player action in isolation risks producing events with no overall narrative shape.

Riedl and Bulitko frame this problem as a balance between coherence and player agency [53]. A system that scripts every outcome keeps the story coherent but removes the player’s ability to affect it. A system that relies entirely on emergent agent behaviour risks producing disconnected events rather than a comprehensible plot. The *drama manager* is proposed as a resolution: a component that observes the state of an interactive story and intervenes to keep it within author-defined bounds, without removing the player’s immediate agency.

2.4.2 Two-Tier Architectures and the Authoring Bottleneck

A drama manager typically operates alongside a separate layer of local, reactive behaviour, forming a two-tier architecture. The first tier consists of individual agents, responsible for moment-to-moment reactions. The second is the drama manager, which does not control any agent directly, but observes the global state and intervenes at a higher level, for example by introducing an event or moving the story to a new phase.

Mateas and Stern’s *Façade* (2005) is the canonical example of this model [38]: character agents handle immediate reactive behaviour, while a global drama manager sequences high-level story beats. The project showed that a two-tier architecture can produce responsive, narrative-driven agents. It also exposed the limitation of the approach, its authoring burden. Mateas and Stern report that authoring the narrative and behaviour logic of their system required multiple person-years of specialised programming effort [38]. The architecture works conceptually, but expressing it requires programming expertise.

This authoring bottleneck is the gap this thesis addresses. The two-tier architecture suits narrative games, but designers need an accessible tool to author it. Domain-specific languages, introduced next, are a general software engineering answer to this kind of authoring burden, and Section 4.3 returns to the two-tier architecture.

2.5 Domain-Specific Languages

General-purpose languages require domain experts to translate their conceptual models into low-level code, a translation often called the semantic gap. Mernik, Heering, and Sloane define a Domain-Specific Language (DSL) as a language tailored to an application domain to bridge this gap [40]. Its abstractions and notations are drawn from the domain’s vocabulary rather than from general-purpose programming constructs. A DSL program can therefore read more like a description of the problem than an implementation of it. This lets domain experts work more productively, and reduces the likelihood of implementation errors.

Van Deursen, Klint, and Visser distinguish two implementation strategies [58]. A standalone DSL has its own compiler or interpreter, and is free to adopt whatever notation fits the domain. An embedded DSL lives inside a general-purpose host language, reusing its parser and approximating domain notation through user-defined operators or functions. Mernik, Heering, and Sloane describe the same trade-off [40]. An embedded DSL inherits the power and tooling of its host, but its syntax is only as flexible as the host allows. A standalone DSL can match the domain’s notation and report errors in the domain’s terms, but requires building and maintaining a full language implementation.

A standalone DSL does not need to generate machine code. Mernik, Heering, and Sloane list source-to-source translation among the ways to implement a DSL: a program in the new language is translated into a program in an existing one, which then compiles or interprets it as usual [40]. A language implemented this way, commonly through a transpiler, keeps the notational freedom of a standalone DSL, while inheriting a runtime, a toolchain, and an execution model it does not have to build.

Visual programming extends the accessibility principle of DSLs by replacing textual syntax with a graphical representation. DeLozier and Shey show empirically that visual programming environments lower the barrier to entry for novice programmers, compared with text-based alternatives [13]. A visual tool rules out syntax errors by restricting interaction to valid graphical connections. It also shows control flow and program state on screen, rather than requiring a reader to reconstruct them from code. Game designers, the audience of this thesis, typically combine expertise in game flow and narrative pacing with little software engineering training (Chapter 1). This is the gap that both a domain-specific notation and its visual representation aim to close.

2.6 Integration Architecture: VEsNA and Godot

An agent’s beliefs and actions must ultimately connect to a running game world, and this connection raises problems of its own, which no authoring language solves. This section introduces the architectural conflict between a real-time engine and a BDI agent, Godot Engine, the environment this thesis assumes, and VEsNA, the framework that resolves the conflict for it.

2.6.1 The Real-Time Engine Environment and the Impedance Mismatch

A real-time game engine, such as Unity or Unreal Engine, is built around a synchronous, fixed-frequency update loop: rendering, physics, and audio are recomputed on every tick, typically dozens of times per second. A BDI reasoning cycle (Section 2.2.2) follows no such schedule. Its duration varies with the agent’s beliefs and plans, and Jason’s implementation of it (Section 2.3.5) is asynchronous by design. Mariani and Omicini identify connecting a general-purpose multi-agent system to a real-time game engine as an open research challenge, because of this mismatch [35]. Forcing reasoning to run within the engine’s tick budget risks either cutting deliberation short or stalling the rendering loop while it waits. Running the two fully decoupled, with no synchronisation, risks missed perceptions and actions applied out of order. Some bridging mechanism is therefore needed, whichever engine and agent platform are involved.

Godot Engine is the engine this thesis targets [23]. It is open source, and VEsNA has adopted it. Godot organises a running game as a hierarchical scene tree, in which entities are nodes that communicate mainly through signals: when a notable event occurs, a node emits a signal, and nodes listening for it react. This keeps a Godot project modular and event-driven, and is the mechanism a bridge to an external reasoning system plugs into.

2.6.2 VEsNA: Goals and Architecture

VEsNA (Virtual Environments via Natural language Agents) is a framework for building embodied BDI agents, developed by Gatti, Mascardi, and colleagues as an extension of Jason [18]. Its scope is broader than the impedance mismatch: a VEsNA agent has a situated body, reasons spatially about its surroundings, communicates with users and other agents in natural language, and behaves according to a simulated personality and mood [18]. The first published version targeted factory automation, letting a user without a programming background populate and query a simulated factory through natural-language commands, on a stack combining Dialogflow, JaCaMo, and Unity [21, 20]. VEsNA has since gener-

alised beyond that domain, and its current implementation runs on Jason on the agent side and targets Godot on the environment side [18, 16].

The part of VEsNA relevant to this thesis is how it resolves the impedance mismatch. A VEsNA agent is split into a mind and a body. The mind runs the BDI reasoning cycle; the body executes the mind's directives in the game engine and reports events and action outcomes back to it. The two communicate asynchronously, through JSON messages over a WebSocket connection, rather than within the engine's tick [18]. Through this channel, the environment adds beliefs to an agent's mind, triggering its plans, and the agent sends commands that act on the environment. Neither the reasoning cycle nor the rendering loop waits on the other: synchronisation happens explicitly, through messages, rather than implicitly, through a shared call stack.

VEsNA embodies an existing BDI agent in a live simulation. It does not address how a non-programmer authors that agent's logic, the problem this thesis addresses.

Chapter 3

Related Work

This chapter evaluates existing tools and methodologies against the combination of capabilities the problem of this thesis requires, to identify the gap that remains.

We organise the evaluation around three capabilities introduced in Chapter 2: **accessibility**, whether someone without a software engineering background can author with a tool; **deliberative reasoning**, whether a tool represents an agent’s competing desires, and its commitment to one of them, as first-class elements rather than only reactive control flow; and **engine embodiment**, whether it connects an agent’s logic to a live, real-time simulation. A further concern, narrative orchestration, the coordination of several agents into one story described in Section 2.4, cuts across all three, and we note it wherever a tool addresses it.

Section 3.1 examines tools that prioritise accessibility, such as visual scripting systems and narrative design tools, and asks how far they reach deliberative reasoning. Section 3.2 examines BDI modelling languages, which prioritise deliberative reasoning, and asks the same of accessibility. Section 3.3 examines middleware that prioritises engine embodiment, and asks whether any of it addresses how an agent’s logic is authored. Section 3.4 maps the coverage across all three capabilities and states which combination remains unaddressed.

3.1 Model-Driven Game Development and Authoring Tools

The game industry increasingly relies on high-level authoring tools to improve productivity. Furtado and Santos argue that game development benefits from visual modelling and code generation [17]. This approach, Model-Driven Game Development (MDGD), shifts

the focus from writing code to defining domain-specific models [65]. MDGD addresses accessibility: a domain expert defines a model directly, rather than translating it into code by hand. We examine three kinds of MDGD tool: visual scripting for reactive game logic, textual and visual tools for narrative content, and an authoring tool for narrative orchestration built around a director agent.

3.1.1 Visual Scripting and Reactive Logic

Visual scripting systems apply MDGD in commercial engines, and Unreal Engine’s Blueprint system is a prominent example [63]. A designer builds behaviour by connecting nodes on a canvas, which rules out syntax errors by construction and shows execution flow on screen.

PlayMaker, a third-party tool for Unity, applies the same paradigm more narrowly: its visual editor is built around the finite state machine model of Section 2.1.1, with states, actions, and events, rather than a general node graph [24]. Shipped titles using it range from *Hearthstone* (2014) to *Hollow Knight* (2017).

Both tools author the reactive paradigms of Section 2.1: Blueprints in the event-driven scripting tradition, PlayMaker in the finite state machine tradition.

3.1.2 Narrative Domain-Specific Languages

Narrative content is the target of a mature line of accessible authoring tools, both textual and visual. Inkle’s Ink is a text-based language for interactive narrative [25]. A script is organised into named knots and stitches connected by diverts, with a lightweight syntax for inline branching choices, and variables tracked as the narrative progresses. Titles using it include *80 Days* (2014) and *Heaven’s Vault* (2019).

Yarn Spinner takes a more minimal approach [55]. It originated in the production of *Night in the Woods* (2017), where a small team needed a dialogue system usable by a writer who was not primarily a programmer. Its scope is deliberately narrow: a script can only deliver a line of dialogue, present a set of options, or issue a command for the host game to interpret, and code written around it handles everything else. It has since been adopted in other games, such as *A Short Hike* (2019) and *Dredge* (2023).

articy:draft approaches the same problem visually [4]. Storylines, characters, and variables are composed in a node-based “Flow” view, which serves a writing team as a shared content database, with export pipelines into Unity and Unreal. Titles produced with it include *Disco Elysium* (2019) and *Hogwarts Legacy* (2023).

The three tools share one underlying model. Whether written as Ink knots, Yarn Spinner nodes, or an articy:draft flow, a script represents branching text, conditions, and state

variables: what is said, and when. How a character behaves is left to a separately authored engine layer. Together with visual scripting, these tools achieve accessibility twice: for reactive game logic, and for narrative content.

3.1.3 Interactive Drama Authoring: Scribe

A third kind of accessible tool addresses neither reactive logic nor dialogue, but the narrative orchestration a two-tier drama management architecture requires. Medler and Magerko’s Scribe is built for this purpose [39]. It was developed alongside a training system in which a director agent manages a story world, responding to authored content and to a trainee’s actions in a 3D environment. Its motivation is accessibility: subject matter experts, typically without programming experience, need to author the director’s story content without a programmer encoding it. An author works in three modes: placing the objects, points, and zones a story references on a map; composing plot points from preconditions, events, and actions that the director consults at runtime; and querying the director to see how it would respond to a given story state, before running the story.

Scribe’s authors state what the tool does not do. It assumes NPCs are already goal-based and directable, and, in their words, “has the ability to work with NPC behaviors but does not create them”, leaving that task to separate systems such as ABL [37], the language used in *Façade*. Scribe is therefore the one system surveyed here that combines accessibility with narrative orchestration by design, while drawing its boundary at deliberative reasoning.

3.2 Methodologies and Modelling Languages for BDI Development

The multi-agent systems community recognises that authoring raw BDI logic is complex, as the tools of Section 3.1 recognise for game logic. Researchers have responded with graphical methodologies and model-driven tools, which let an author manipulate diagrams or generate code rather than write agent logic line by line. By design, these tools do not change the vocabulary an author works in: the concepts remain beliefs, goals, plans, and the other agent-theoretic terms of Section 2.2.

Padgham and Winikoff’s Prometheus is a software engineering methodology for designing intelligent agents [45]. It provides a top-down process expressed through graphical artifacts, from an overview of the system’s goals and capabilities, through diagrams of agent types and their interactions, to a detailed internal design. A system architect uses Prometheus to plan a multi-agent system before implementing it. Its diagrams, however, show goals, beliefs, and capabilities: the vocabulary of a raw BDI implementation, arranged more

legibly but not translated into other terms.

Kardas, Tezel, and Challenger’s DSML4BDI takes a different route [31]. An author builds a visual model of an agent, connecting nodes that represent beliefs, goals, and plans, and the tool generates AgentSpeak code for Jason from it. The authors report that this can reduce the time needed to develop a BDI agent system by up to two thirds. The simplification is procedural, since the author no longer writes Jason syntax, but not terminological: the nodes are still labelled belief, goal, and plan.

Prometheus and DSML4BDI belong to a larger family of approaches developed independently since the early 2000s. Some structure the design process, others generate code from a model. Among the methodologies are Gaia, which organises a system around roles and their interactions [61], and Tropos, which grounds agent design in early requirements analysis [10]. A recent survey by Abdalla and Mishra finds that the number of such approaches, and the interest in applying them, continues to grow [1].

This proliferation became a problem in its own right. Beydoun, Low, Henderson-Sellers, and colleagues address it with FAML, a generic metamodel that reconciles the concepts of competing methodologies into one common foundation [6]. FAML was built by comparing and unifying the concepts of seven methodologies, Gaia and Tropos among them, and is intended for methodology designers rather than for the implementation of individual systems. That so many independently developed approaches needed reconciling indicates how large and mature this tradition is. Chapter 8 uses FAML as the reference for evaluating Regia.

Across these methods and tools, the simplification is real: a diagram is easier to construct correctly than hand-written AgentSpeak, and a shared metamodel reduces duplicated effort. That usability, however, is assessed for a developer already fluent in beliefs, goals, and plans, not for a domain expert who works in a different vocabulary. Accessibility outside the BDI vocabulary, and narrative orchestration across several agents, remain unaddressed.

3.3 Agent-Engine Integration Middleware

Sections 3.1 and 3.2 showed that accessible authoring tools do not reach deliberative reasoning, and that BDI modelling tools do not reach authors outside the BDI vocabulary. Once an agent’s reasoning is written, it must also run inside a real-time game engine. Section 2.6.1 described the conflict this raises between an asynchronous reasoning cycle and a synchronous rendering loop, which Mariani and Omicini identify as an open research challenge [35]. Several frameworks have since addressed it, each with a different strategy.

Angilica, Ianni, and Pacenza’s ThinkEngine keeps reasoning external to Unity and moves its computational load outside the main game loop [3]. A ThinkEngine “Brain”, an Answer

Set Programming (ASP) module, a declarative paradigm in which a problem is described by logic rules and its solutions are computed as the answer sets of the program [11]. A Brain can be attached to any non-player character. Configuring one means choosing the sensors that feed it, the events that trigger its reasoning, the ASP program that encodes its decision strategy, and the actuators that carry out its conclusions. The result is a hybrid architecture: declarative reasoning decides at a strategic level, while Unity’s engine loop handles moment-to-moment execution.

VEsNA, described in Section 2.6.2, follows a related strategy for BDI reasoning [18]: an agent’s mind runs in Jason, outside the engine, and connects to it asynchronously over a WebSocket bridge. In its current implementation, the engine is Godot rather than Unity.

Poli takes a different approach, avoiding the network boundary both frameworks rely on [49]. The work proposes a new BDI language, based on Prolog and inspired by Jason, executed by a Prolog interpreter running inside Unity. An Artifact abstraction models the scene’s environment, and a communication layer connects agents and artifacts, both following the vocabulary of CArtAgO. This removes the boundary rather than bridging it, at the cost of building and maintaining a new reasoning language.

Pascucci’s Synopsis generalises the external bridge into a dedicated middleware component, instead of a point-to-point connection between the two runtimes [47]. Each entity is split into a mind, a JaCaMo agent reasoning over a CArtAgO artifact, and a body, the corresponding Unity GameObject. The middleware mirrors this split with the actor model of the Play framework [48]: each mind and body opens its own WebSocket connection, which Play wraps in a dedicated actor. Every message therefore carries the identity of its sender and recipient, and no separate dispatcher is needed. The strategy is close to that of VEsNA, but generalised to connect any multi-agent system to any game engine, at the cost of the bookkeeping the mirrored actors require.

The four frameworks show that the mismatch between reasoning and rendering can be solved through different strategies: external declarative reasoning bridged to a game loop, an external BDI mind bridged over a network, a BDI interpreter embedded in the engine, and a middleware independent of any reasoning language. Engine embodiment is therefore the best covered of the three capabilities. None of the four frameworks, however, addresses how the logic they embody is authored: a developer still writes ASP, Prolog, or AgentSpeak in a text editor, with the limits on accessibility found in Section 3.2.

3.4 Summary

Each capability on its own is served by at least one mature line of work. Visual scripting achieves accessibility for reactive game logic, and textual and visual tools achieve it for

narrative content. A large and growing tradition of BDI methodologies and modelling languages serves deliberative reasoning. At least four frameworks solve engine embodiment, through different strategies. No surveyed tool, however, combines accessibility with deliberative reasoning.

Every framework that embodies BDI reasoning in a real-time engine still requires its author to write that reasoning in ASP, Prolog, or AgentSpeak, with the fluency this demands. Every tool that lets a non-programmer author content, whether game logic or branching narrative, offers no way to describe how an agent deliberates. Scribe comes closest, combining accessibility with narrative orchestration, but its authors explicitly leave the behaviour of each character to other systems. No surveyed system, in other words, treats a scenario involving several independently reasoning agents as its own authorial unit, accessible to someone without a programming background, down to how each agent deliberates. Chapter 4 takes up this combination.

Chapter 4

Requirements and Analysis

Chapter 1 described the origin of this project and the interest of Untold Games in the Belief-Desire-Intention model. In this chapter we describe how that interest became a set of requirements for a new language: who Regia serves, what that audience needs from its syntax and vocabulary, and how our collaboration with Untold Games reshaped what a Regia program describes. The current implementation does not meet every requirement in full. Where it falls short, we say so, and Chapter 9 returns to it.

Section 4.1 describes the audience of Regia and how we gathered its requirements. The following five sections derive one requirement each: an accessible syntax and vocabulary (Section 4.2); a narrative-centred model (Section 4.3); dynamically assigned behaviour (Section 4.4); a visual representation of the narrative (Section 4.5); and modularity and reuse (Section 4.6).

4.1 Stakeholders and Methodology

Regia is designed for game designers, not programmers, and this choice shapes nearly every requirement in this chapter. Chapter 1 described the friction between the two roles. Here we focus on one asymmetry it produces. A designer’s description of a narrative is expressive but informal, leaving implementation details for the programmer to infer or to ask about. The implementation that results is precise, but often illegible to the designer who wrote that description, so they cannot check for themselves whether their intent survived the translation. Regia is positioned between the two: formal enough to be executed, and readable enough for a designer to write and check without a programmer’s help. It eases this friction without removing either role from the process.

We gathered requirements informally, over a series of meetings with our co-supervisor

there, Fabio Casale. The meetings fell into two loosely sequential phases. In the earlier meetings, before any working language existed, we compared our emerging conception of the language with what the designers expected such a tool to let them express. In the later meetings, we walked Fabio through example scripts of specific constructs and collected their reactions. These reactions often showed that a construct had not been understood as intended. We then renamed the element with a term closer to the designers' vocabulary, or restructured part of the language, as the following sections describe. We accepted feedback where it improved usability for designers, but kept the underlying model aligned with the agent-centric view of the system, since this alignment keeps the language translatable into AgentSpeak. Where a request would have broken that translation, we addressed its underlying concern in a different way.

4.2 Syntax and Accessibility

The audience described in Section 4.1 sets the first requirement, R1: a designer with no programming background must be able to read a Regia script, and ideally write simple ones, without first learning a dedicated notation or unfamiliar technical terms. Chapter 3 showed that accessibility of this kind exists for reactive game logic and narrative content (Section 3.1), but not for BDI authoring (Section 3.2). R1 resolved into two kinds of choice: how a construct is written, its notation, and what a construct is called, its vocabulary.

4.2.1 Notation

The notational requirement resolved into three choices: capitalised keywords in place of symbolic operators, little punctuation, and no variables.

Placing a Regia plan next to the AgentSpeak it compiles to shows the effect of these choices. Listing 4.1 shows the `digestion_check` plan of the running example introduced in Chapter 5, and Listing 4.2 an equivalent AgentSpeak plan.

```
1 WHEN digestion_check:  
2     IF is_hungry:  
3         DO eat_food.  
4         DO FORGET(is_hungry).  
5         DO PRINT("A guest finishes their meal.").
```

Listing 4.1: The `digestion_check` plan of the running example, in Regia.

```

1 +digestion_check : is_hungry <-
2   eat_food;
3   -is_hungry;
4   .print("A guest finishes their meal.").

```

Listing 4.2: An equivalent plan in AgentSpeak.

Regia has no equivalent of the AgentSpeak symbols introduced in Chapter 2: `+` and `-` for belief addition and removal, `:` and `<-` separating a plan’s trigger, context, and body, and `;` separating statements. It states the same structure with capitalised English words, **WHEN**, **IF**, **DO**, and **FORGET**, each mapped onto one of these symbols. A reader can therefore follow a plan without first learning what each symbol means. Capitalisation sets a keyword apart from the Facts, Actions, and Events around it, without further punctuation.

The comparison above does not illustrate the third choice, since neither plan uses a variable. AgentSpeak, following Prolog, unifies variables inside a plan’s context: a context such as `enemy(X) & health(X, H) & H < 10` binds `X` and `H` to whatever values satisfy the belief base when the plan is considered. Regia has no equivalent. A Fact reference such as `hates(host)`, described in Section 5.2, always takes a concrete, named argument. Avoiding variables entirely was itself a stated requirement, since unification has no counterpart in the natural-language descriptions designers already write, and requires a logic-programming background to build an intuition for.

4.2.2 Vocabulary

A simple concept can still be hidden behind an unfamiliar word, leaving a designer to learn new terminology for something they already understand. This reasoning guided several of Regia’s vocabulary choices, and **FACT** is the clearest instance of it.

The standard term in the BDI literature, introduced in Chapter 2, is *belief*. The word carries doxastic connotations from the theory BDI is built on, which mean nothing to a designer and add jargon to a simple concept: a statement an agent holds to be true about the game world. A designer already uses *fact* informally for such a statement, without needing the technical, agent-theoretic sense of “belief”. The underlying concept is unchanged from AgentSpeak; only its label differs.

4.3 From Agents to Narratives

The most significant change in Regia’s design concerned its point of view: what a program is about. The change took three steps, each moving the centre of a description further from the individual agent and towards the scenario it takes part in.

The starting point is AgentSpeak, introduced in Chapter 2. There, the unit of description is a single agent: a program is that agent’s set of plans, and no construct describes a shared scenario. A story spanning several agents exists only in the programmer’s head. Early in the collaboration, this proved a poor fit for how the designers of Untold Games worked: they did not describe agents in isolation, but scenarios, with several participants and a flow of their own. The first prototypes of Regia were syntactic simplifications of AgentSpeak, and shared this limitation. A second version therefore introduced a construct for a scenario, **STORY**, which wrapped the participating agents and divided the scenario into **PHASES**, in which each agent could behave differently.

Having a scenario construct, however, did not make the scenario the *subject* of a description. In this second version, the agents were still described individually inside the construct, each carrying its own reactions. The scenario was a container, not the thing being described. The final version, presented in Chapter 5, makes the scenario itself the subject: an agent’s behaviour becomes something a narrative draws on, rather than something written into it.

4.3.1 Snippets

The gap became clear when we compared the designers’ documents with what the second version asked an author to write. Figure 4.1 shows one such document, which the designers called a *snippet*: a self-contained scenario in which a Citizen asks the Player to fetch an item, laid out as a branching flow of dialogue, conditions, timers, and consequences.

Listing 4.3 shows one phase of the same quest in the second version of Regia. Its syntax differs from the final language; what matters is how the description is organised.

```
1 DURING asking:
2
3   TRANSITION TO searching WHEN player_accepts IF citizen_is_known.
4   TRANSITION TO failed    WHEN player_rejects.
5
6   AGENT Citizen:
7     WHEN player_accepts IF citizen_is_known:
8       DO acknowledge.
```

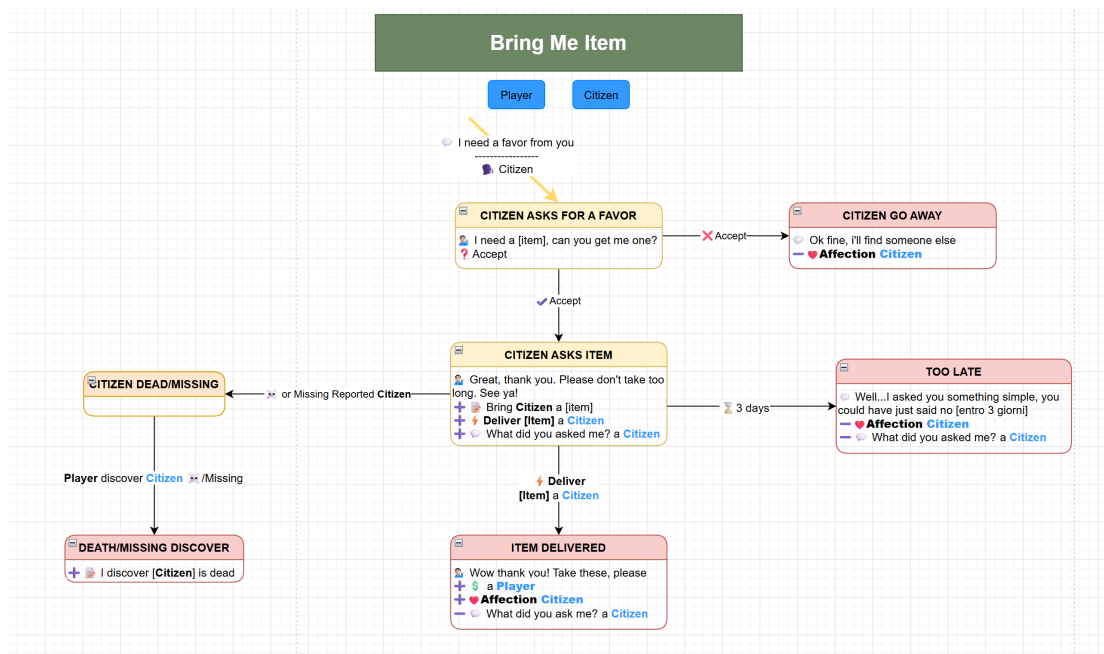


Figure 4.1: A “Bring Me Item” snippet, one of the design documents shared by Untold Games during early requirements gathering.

```

9      WHEN player_rejects:
10         DO find_someone_else.
11
12     AGENT Player:
13         WHEN player_accepts IF NOT is_busy:
14             DO acknowledge.
15         WHEN player_rejects:
16             DO explore.

```

Listing 4.3: One phase of the “Bring Me Item” quest in the second version of Regia.

The scenario construct divides the quest into phases, but within each phase every participant’s reactions are written separately, one block per agent. This has two consequences. First, following the scenario’s flow, what happens when the Player accepts and what happens next, requires reading several per-agent blocks and reassembling them mentally, since no single place describes the flow. Second, behaviour relevant in more than one phase is written again in each: in the full source of this excerpt, the Citizen’s reaction to being abandoned appears in three separate phases, three copies of one participant reacting to one thing.

The snippet and the code are therefore organised in opposite ways. The snippet follows

the scenario: a favour is asked, accepted or refused, and an item delivered in time or not. The Citizen and the Player are participants in that flow, not its subject. The code follows the agents, and leaves the flow implicit, to be reconstructed from every agent’s fragments across every phase. Although a construct for the scenario already existed, the language still asked an author to take a situation apart into per-agent pieces in order to write it down.

4.3.2 Making the Narrative Central

These observations set the second requirement, R2: a Regia program must describe a scenario as a whole, with each participant’s behaviour defined once and attached to the part that participant plays. Meeting it meant inverting what the language put at its centre. The scenario construct became the primary one, now called a Plot. The part a participant plays within it is called a Role, and the behaviour attached to a Role, defined once and reusable across phases, is bundled into a Playbook. Chapter 5 describes all three. Together, they let an author write a situation down as a situation, without decomposing it into per-agent, per-phase fragments.

This split closely mirrors the two-tier architecture of the interactive narrative literature, described in Section 2.4.2, in which a drama manager sequences story beats while character agents handle moment-to-moment behaviour, as in Mateas and Stern’s *Façade* [38]. We did not set out to reproduce that architecture: we noticed the resemblance only once the split between Plot and Playbook was in place.

Treating a scenario as a self-contained unit has one further consequence. A larger story, and ultimately a game, can be built by composing such units: one scenario giving way to another, or opening a smaller one nested inside itself, so that the whole forms a hierarchy of scenarios rather than one flat script. Regia realises this hierarchy through Subplots, also described in Chapter 5.

4.4 Dynamic Behaviour

A Playbook’s attachment to a Role is not fixed for the whole run of a narrative: what a participant can react to often needs to change as the situation develops. This is a different requirement from ordinary branching. A reactive rule that chooses between two outcomes on a condition already works within one fixed set of behaviours. What designers asked for was a change in *which* behaviours are available to be considered at all.

A hypothetical guard illustrates the distinction. On routine patrol, the guard reacts to a handful of mundane situations, such as noticing an open door or greeting a passer-by.

Once the guard suspects an intruder, a different set of reactions becomes relevant: calling for backup, pursuing, searching a room. None of these is meaningful, or perhaps safe to leave active, while the guard is on its rounds. Branching alone could encode this, with conditions gating every reaction, but every reaction would remain permanently present and reasoned about together. The designers asked instead for a way to state that a whole bundle of reactions becomes available, or stops being available, as the narrative moves a participant from one situation to another. This is the third requirement, R3.

Jason already provides the mechanism: an agent can add and remove plans from its own plan library at runtime, through the `.add_plan` and `.remove_plan` internal actions [8]. Using it as the primary way to organise behaviour is, however, uncommon. The idiomatic pattern is the opposite: an agent holds its full repertoire of plans from the start, and each plan’s context decides whether it fires, matching against the agent’s current beliefs. The plan set itself does not change.

Our collaborators at Untold Games requested this pattern repeatedly, and Regia makes it explicit and central through the Playbook: a named, reusable bundle of reactive behaviour that a narrative attaches to a Role at one point and detaches at another (Sections 5.3 and 5.4). In the compiled AgentSpeak, Regia in fact follows the idiomatic pattern, including every Playbook an agent may receive and gating its plans with a belief (Section 6.1.4). The author, however, sees behaviour attached and detached, which matches how the designers describe it. In the guard example, patrol and alert behaviour become two Playbooks, and the narrative, not the guard, decides which one the guard has. A Playbook can also be attached to more than one Role, or reused across narratives; Section 4.6 returns to this second dimension, reuse rather than dynamism.

4.5 Visualisation

A further requirement concerns how a narrative is seen while it is being written. The snippet of Figure 4.1 is not prose. It is a diagram: branches, conditions, and named outcome states connected by arrows, close in spirit to a state machine. Before we proposed any visual representation, the designers had already chosen a diagram to lay out the kind of structure Regia’s Plots describe.

This observation bears on R1. Section 4.2 grounded accessibility in a notation that a reader without programming experience can follow. A textual notation, however readable, remains text, and designers who already think and communicate in diagrams are not fully served by it. A visual representation of the narrative is therefore an extension of R1 rather than an optional addition, and forms the fourth requirement, R4.

Pairing an executable notation with a visual representation of the same logic has precedent

in game development. Behaviour trees (Section 2.1.2) [26] and Unreal Engine’s Blueprints (Section 3.1.1) [63] both let a designer work with a system’s logic visually, rather than only through its textual form.

The Visual Editor built in response, described in Section 6.2, draws a narrative’s structure: its Phases and the transitions between them, consistent with the narrative-centred model of R2. It gives no visual form to the reactive logic inside a Playbook. R4 asked for a way to see a narrative as our collaborators already sketched one, and the editor addresses that part of the language.

4.6 Modularity and Reuse

The fifth requirement, R5, differs in character from the others. R1 to R4 bear on the language’s semantics or on the experience of writing a script, while a designer writing a single scenario would rarely notice the absence of modularity and reuse. They matter because they shape the language’s underlying structure, which is harder to change later than a surface feature. We therefore made several of the decisions below early, alongside the shift of Section 4.3, although we discuss them last.

4.6.1 Practical Hygiene

The most direct form of R5 is practical. As Plots and Playbooks accumulate, a single file becomes hard to navigate. `IMPORT`, described in Section 5.1, lets an author split a project across files while still referencing declarations made elsewhere. Documentation comments, described in the same section, let an author leave a short note on a Plot or a Playbook for a later reader, programmer or designer, without leaving the language.

4.6.2 Composability

A behaviour tree is built from smaller, independently authored subtrees, which can be developed, tested, and reused on their own before being combined (Section 2.1.2). Reuse across projects, and not only organisation within one, depends on this property.

Regia’s Roles, Playbooks, and Subplots were shaped with the same goal. A Role separates the parts a narrative defines from the concrete agents that fill them at runtime, so that a Plot can be reused with a different cast, or in a different game, without changes. A Playbook is a self-contained bundle of behaviour, so that it can be attached to several Roles and carried between projects. Subplots apply the same principle to Plots: an author

builds a Plot from smaller, independently authored Plots, as a behaviour tree is built from subtrees. A game’s narrative can then be represented as a tree of Plots and Subplots.

Composability has a cost. Once Playbooks and Plots can be freely combined, two independently authored pieces of behaviour, each reasonable on its own, may be active on the same agent or narrative at once and disagree. **PRIORITY**, described in Section 5.3.5, was conceived for this case: an automatic arbitration between concurrently active Playbooks, or between concurrently running Plots and Subplots, so that an author can reuse a piece of behaviour without anticipating every other piece it may be combined with. As that section explains, the target platform does not yet enforce it.

4.7 Summary

Table 4.1 lists the five requirements derived in this chapter and the parts of Regia that address them. Chapter 5 presents the resulting language in full.

ID	Requirement	Section	Addressed by
R1	A designer can read Regia, and write simple scripts, without learning a symbolic notation or technical terms	4.2	Keywords in place of symbols; no variables; FACT
R2	A program describes a scenario as a whole, with each participant’s behaviour defined once and attached to its part	4.3	Plots, Roles, Playbooks, Subplots
R3	Bundles of behaviour become available and unavailable as the narrative progresses	4.4	ASSIGN , UNASSIGN
R4	A visual representation of a narrative’s structure	4.5	The Visual Editor
R5	Projects split across files; behaviour and narratives composed and reused	4.6	IMPORT , doc comments, Roles, Playbooks, Subplots, PRIORITY

Table 4.1: The requirements derived from the collaboration with Untold Games, and the parts of Regia that address them.

Chapter 5

Language Design

This chapter presents the syntax of Regia’s final design: each construct, its role, and its translation into AgentSpeak. Chapter 6 describes how the transpiler implements this translation.

We give the grammar of each construct in BNF, and illustrate every construct with one running example. Listing 5.1 shows its complete source: a royal banquet in which a guest, insulted by the host, may challenge them to a duel that interrupts the feast. We refer to fragments of this listing by line range.

```
1  # Vocabulary Declarations
2
3  # Actions used by Playbooks
4  ACTION toast(target).
5  ACTION eat_food.
6  ACTION complain.
7
8  # Actions used by the Director (WORLD DO and Role DO)
9  ACTION engine.security.summon_guards AS call_guards.
10 ACTION lock_doors.
11 ACTION play_music.
12 ACTION serve_wine.
13 ACTION stop_music.
14
15 # Events triggering Playbook and Plot WHEN blocks
16 EVENT food_served.
17 EVENT insult_heard.
18 EVENT digestion_check.
19 EVENT assassin_spotted.
```

```

20 EVENT insult_thrown.
21 EVENT duel_resolved.
22
23 # Facts used in Playbook IF conditions
24 FACT is_hungry.
25 FACT is_drunk.
26 FACT hates(person).
27
28 # Playbooks
29
30 #@desc: Reactive behaviour for guests mingling and eating at the
banquet.
31 PLAYBOOK NobleSocializing:
32
33   WHEN insult_heard PRIORITY 5 TEMPER sympathy(-0.5) EFFECTS
    aggressiveness(0.2):
34     IF NOT is_drunk AND hates(host):
35       DO complain.
36       SIGNAL insult_thrown.
37     ELSE:
38       DO toast(host).
39
40   WHEN digestion_check:
41     IF is_hungry:
42       DO eat_food.
43       DO FORGET(is_hungry).
44       DO PRINT("A guest finishes their meal.").
45
46   WHEN food_served:
47     DO PRINT("A guest notices food arriving.").
48     IF (is_hungry OR is_drunk) AND NOT hates(host):
49       DO eat_food.
50     ELSE:
51       DO complain.
52
53 # Plots
54
55 #@desc: The main narrative flow of the Royal Banquet.
56 PLOT RoyalBanquet.
57
58   PHASE reception INITIAL.
59   PHASE feast.
60   PHASE duel_interruption.
61
62   ROLE Host.
63   ROLE Guest.
64

```

```

65  DURING PLOT:
66      WHEN assassin_spotted PRIORITY 9:
67          WORLD DO lock_doors.
68          Host DO call_guards.
69          END PLOT.
70
71  DURING reception:
72      ON ENTER:
73          WORLD DO play_music.
74          ASSIGN NobleSocializing TO Guest.
75
76      WHEN food_served:
77          TRANSITION TO feast.
78
79  DURING feast:
80      ON ENTER:
81          WORLD DO serve_wine.
82
83      WHEN ROLE Guest SIGNALS insult_thrown:
84          WORLD DO stop_music.
85          START SUBPLOT HonorDuel MAPPING Guest TO Challenger.
86          TRANSITION TO duel_interruption.
87
88      ON EXIT:
89          UNASSIGN NobleSocializing FROM Guest.
90
91  DURING duel_interruption:
92      WHEN SUBPLOT HonorDuel ENDS:
93          WORLD DO PRINT("The duel is settled. The banquet is ruined.
94              ").
95          END PLOT.
96
97  #@desc: A duel fought by an insulted guest.
98  PLOT HonorDuel.
99
100  PHASE combat INITIAL.
101
102  ROLE Challenger.
103
104  DURING combat:
105      WHEN duel_resolved:
106          WORLD DO PRINT("The duel is over.").
107          END PLOT.

```

Listing 5.1: The running example: a royal banquet interrupted by a duel.

5.1 Program Definition

```
1  <program>          ::= <import-stmt>+ <item>*
2                        | <item>+
3
4  <import-stmt>      ::= 'IMPORT' STRING '.'
```

A Regia program consists of import statements followed by items. An import statement includes the contents of another Regia source file, given by its path as a string. Imports let an author split a project into modules and reuse a module across programs.

Imports must appear at the top of the file: the grammar enforces this order, and the compiler rejects a program in which an import follows an item. The running example is self-contained and declares no import. A larger project would typically move its vocabulary into a shared file imported by several Plots:

```
1  # banquet_vocab.regia
2  FACT is_hungry.
3  ACTION eat_food.
4  ...
5
6  # royal_banquet.regia
7  IMPORT "banquet_vocab.regia".
8
9  PLAYBOOK NobleSocializing:
10     ...
```

Once resolved, every symbol declared in `banquet_vocab.regia` is available in `royal_banquet.regia`, as if both files were one. Section 6.1.2 describes this resolution.

```
1  <item>              ::= <element-decl>
2                        | <playbook-def>
3                        | <plot-def>
```

After the imports come the items, the building blocks of a Regia program. There are three kinds:

- **Element Declarations:** declarations of Facts, Events, and Actions, the shared vocabulary described in Section 5.2.
- **Playbook Definitions:** definitions of Playbooks, reusable bundles of reactive plans, described in Section 5.3.
- **Plot Definitions:** definitions of Plots, narrative state machines, described in Section 5.4.

Listing 5.1 follows this order: a vocabulary block, one Playbook, and two Plots. Comments fall outside the grammar, and we describe them first.

5.1.1 Comments

Regia has two kinds of comment, both introduced by `#`. An ordinary comment, `#` followed by any text, may appear on its own line anywhere in a program and is discarded; the running example uses it to separate its vocabulary, Playbook, and Plot sections. A doc comment, of the form `#@key: value` and optionally continued on further lines with `#-`, attaches a short description to the declaration or definition that follows it:

```
1  #@desc: A duel fought by an insulted guest.
2  PLOT HonorDuel.
```

The running example uses `desc` as its only key, on its Playbook and its two Plots. The grammar does not restrict keys, so an author may add others for a downstream tool, such as the Visual Editor, to read. A doc comment is not discarded: the preprocessor recovers it before parsing, and the compiler reattaches it to the AST node it precedes (Sections 6.1.2 and 6.1.2).

5.2 Vocabulary: Facts, Actions, Events

```
1  <element-decl> ::= <action-decl>
2                  | <event-decl>
3                  | <fact-decl>
```

Facts, Actions, and Events form the shared vocabulary of a Regia program. Playbooks and Plots both reference it, and the compiler requires every element to be declared before its first use. Declarations produce no AgentSpeak: the compiler uses them only for validation. They also give designers one place to list the elements a narrative needs, and to discuss them with programmers. Table 5.1, at the end of this section, shows how references to the vocabulary translate.

Lines 1 through 26 of Listing 5.1 declare the vocabulary of the running example: three Actions available to Playbooks, five Actions meant for the Director, six Events, and three Facts.

5.2.1 Facts

```
1 <fact-decl> ::= 'FACT' ID <param-names>? '.'
```

A Fact declares a named proposition that an agent can believe. A Fact may carry a list of parameter names:

```
1 FACT is_hungry.
2 FACT hates(person).
```

A Fact without parameters, such as `is_hungry`, behaves as a boolean flag: the agent either holds the belief or does not. A parametric Fact, such as `hates(person)`, has a named slot, and the agent may hold several instances of it, each with a different value: a Guest can believe both `hates(host)` and `hates(steward)`. Facts form the epistemic state of an agent, and appear as guards in IF branches (Section 5.3), where they become the context of an AgentSpeak plan.

5.2.2 Actions

```
1 <action-decl> ::= 'ACTION' ID <param-names>? ('AS' ID)? '.'
```

An Action declares a named operation that an agent can execute. Like a Fact, an Action

may carry a list of parameter names. It may also carry an alias, introduced by **AS** after the fully qualified name it stands for.

```
1 ACTION eat_food.  
2 ACTION toast(target).  
3 ACTION engine.security.summon_guards AS call_guards.
```

The alias addresses a practical problem: some underlying AgentSpeak actions are deeply namespaced, and writing their full name in every Playbook and Plot would clutter the script. When an author uses an alias, the compiler substitutes the fully qualified name at emission. In the running example, a Director handler (Section 5.4.2) executes **Host DO call_guards**, and the compiled Host Role file shows the substitution:

```
1 @role__royalbanquet__host__call_guards  
2 #!call_guards <- engine.security.summon_guards.
```

An Action names an interface, not an implementation. Regia states only that the Action exists and, optionally, which parameters it accepts; how it is realised is left to the engine side.

5.2.3 Events

```
1 <event-decl> ::= 'EVENT' ID '.'
```

An Event declares a named trigger that an agent can perceive. Unlike Facts and Actions, Events take no parameters. Events drive plan selection in the BDI cycle: when an Event occurs, the runtime searches for a matching Playbook or Plot handler. In the running example, **food_served** and **digestion_check** trigger the **NobleSocializing** Playbook's **WHEN food_served:** and **WHEN digestion_check:** blocks.

AgentSpeak recognises six kinds of triggering event (Section 2.3.2), among them the addition and deletion of beliefs and goals. A Regia **EVENT** exposes only one of them: the addition of a belief, **+literal**. This restriction follows from the integration system this

thesis targets. The mind-body architecture of VEsNA (Section 2.6.2) delivers almost every perception from an agent’s body to its mind as a belief addition [18], so an author would rarely need to trigger a plan on anything else. Achievement-goal triggers still appear in compiled AgentSpeak, in the `signal_directors` and `switch_phase` plans described later in this chapter, but only in plans the transpiler generates; no Regia construct lets an author write one.

Regia	AgentSpeak
Any FACT , ACTION , or EVENT declaration	No output
IF <code>hates(host):</code>	Plan context : <code>hates(host)</code>
DO <code>toast(host).</code>	Plan body action <code>toast(host)</code>
WHEN <code>food_served:</code>	Plan trigger <code>+food_served</code>

Table 5.1: Vocabulary declarations and references, with their AgentSpeak translation. Declarations are used only for validation.

5.3 Playbooks

A Playbook is a named, reusable bundle of reactive plans. It describes how an agent behaves in a given context, and has no knowledge of Plots or phases: a Playbook can only act on the agent itself, or signal the Director managing the Plot the agent is enrolled in. This isolation keeps Playbooks composable, and allows the same Playbook to be assigned to different agents, or to different Roles across Plots. A Playbook is inactive until a Director enables it with an `ASSIGN` statement (Section 5.4); Section 5.3.1 describes the mechanism.

```
1 <playbook-def> ::= 'PLAYBOOK' ID ':' <pb-when-block>+
```

The running example declares one Playbook, `NobleSocializing` (lines 28 through 51 of Listing 5.1), which we use throughout this section. It contains three `WHEN` blocks, reacting to an overheard insult, to a digestion check, and to food being served. Each Playbook compiles into its own AgentSpeak file, which the file of every Role that may receive it includes (Section 6.1.4). The compiled plans in this section therefore carry the label prefix `@pb__`.

5.3.1 Plans

```
1 <pb-when-block> ::= 'WHEN' ID <priority>? <temper>? ':' <pb-when-body>
```

Each **WHEN** block defines a reactive plan: when the named Event occurs, the agent that perceives it executes the block body. The Event must be a declared **EVENT**. A **WHEN** block may carry a **PRIORITY** annotation (Section 5.3.5); a plan without one defaults to 0.

Each **WHEN** block compiles into one AgentSpeak plan per conditional branch (Section 5.3.3). Every compiled plan carries the corresponding `playbook_active(Name, Director)` belief in its context, so a Playbook produces no behaviour until a Director assigns it. The compiled `digestion_check` plan differs from the plan of Listing 4.2 only by this guard:

```
1 @pb__NobleSocializing__digestion_check__0
2 +digestion_check : playbook_active(noblesocializing, _)
3   & is_hungry <-
4     eat_food;
5     -is_hungry;
6     .print("A guest finishes their meal.").
```

5.3.2 Statements

```
1 <pb-stmt> ::= <do-stmt> | <signal-stmt>
2 <do-stmt> ::= 'DO' <action-name> <arg-list>? '.'
3 <signal-stmt> ::= 'SIGNAL' ID <arg-list>? '.'
```

A plan body is a sequence of statements, of two kinds. **DO** executes an action on the agent itself. **SIGNAL** broadcasts a belief-change event to every Director managing a Plot the agent is enrolled in. **SIGNAL** is the only channel from a Playbook to the narrative layer: to preserve modularity, a Playbook cannot address a Director or a Plot directly, but it can still pass information upward.

The action in a **DO** statement is either a declared **ACTION** or one of seven reserved names that map to AgentSpeak primitives, listed in Table 5.2.

Special Action	AgentSpeak
DO TELL(target, msg).	.send(target, tell, msg)
DO BROADCAST(msg).	.broadcast(tell, msg)
DO ACHIEVE(goal).	!goal
DO BELIEVE(fact).	+fact
DO FORGET(fact).	-fact
DO PRINT(text).	.print(text)
DO WAIT(ms).	.wait(ms)

Table 5.2: The seven special **DO** actions and the AgentSpeak primitives they map to.

The seven keywords fall into four groups. **TELL** and **BROADCAST** send a message to one agent or to every agent, through Jason’s `.send` and `.broadcast` internal actions. **ACHIEVE** posts an achievement goal on the agent’s own agenda. **BELIEVE** and **FORGET** add or remove a belief; the `digestion_check` plan uses **FORGET** so that a guest who has just eaten does not eat again on the next check. **PRINT** and **WAIT** are development aids, which write to the console and pause execution, with no effect on the agent’s beliefs or goals.

The `insult_heard` plan illustrates **SIGNAL**: an insulted, sober Guest who hates the host executes **DO** `complain`, then **SIGNAL** `insult_thrown`. The compiler translates the latter into a call to `signal_directors`, an infrastructure plan generated once per Role file:

```

1 +!signal_directors(PbName, Payload) <-
2   .findall(D, playbook_active(PbName, D), Directors);
3   for ( .member(DirectorId, Directors) ) {
4     .send(DirectorId, untell, Payload);
5     .send(DirectorId, tell, Payload)
6   }.

```

The plan finds every Director that has assigned the signalling Playbook to the agent, and notifies each in turn. The paired `untell` and `tell` force Jason to treat the signal as a new event, even if the same belief was told before and never retracted, so that a repeated insult is not ignored.

5.3.3 Conditional Branches

```
1  <pb-when-body>      ::= <pb-stmt>+  
2                        | <pb-stmt>+ <pb-if-branch>+ <pb-else-branch>  
3                        ?  
4                        | <pb-if-branch>+ <pb-else-branch>?  
4  <pb-if-branch>      ::= 'IF' <condition> ':' <pb-stmt>+  
5  <pb-else-branch>    ::= 'ELSE' ':' <pb-stmt>+
```

A **WHEN** body follows one of three patterns, one for each alternative above.

- In the unconditional pattern, the body is a list of statements, which always executes when the Event occurs.
- In the mixed pattern, unconditional statements precede one or more IF branches. The compiler prepends these statements to the body of every branch, so an author never repeats a common prefix.
- In the conditional pattern, the body is one or more IF branches with no leading statements, optionally closed by an ELSE branch, as in the `insult_heard` plan.

Branches let an author write plans that share a trigger in one place; the compiler still emits each branch as a separate plan. The `food_served` plan illustrates the mixed pattern: the compiler prepends `DO PRINT("A guest notices food arriving.")` to both the IF and the ELSE branch:

```
1  @pb__NobleSocializing__food_served__0  
2  +food_served : playbook_active(noblesocializing, _)  
3    & (is_hungry | is_drunk) & not (hates(host)) <-  
4    .print("A guest notices food arriving.");  
5    eat_food.  
6  
7  @pb__NobleSocializing__food_served__1  
8  +food_served : playbook_active(noblesocializing, _)  
9    & not ((is_hungry | is_drunk) & not (hates(host))) <-  
10   .print("A guest notices food arriving.");  
11   complain.
```

The context of an **ELSE** branch negates the condition of every **IF** branch before it, so it never overlaps with them. **IF** branches are not made exclusive of each other: when two

conditions hold at once, Jason’s default selection function runs the first applicable plan, which is the first matching branch in source order. Either way, one Event runs at most one branch.

5.3.4 Conditions

```

1  <condition>      ::= <condition-and> ( 'OR' <condition-and> )*
2  <condition-and> ::= <condition-atom> ( 'AND' <condition-atom> )*
3  <condition-atom> ::= 'NOT' <condition-atom> | <fact-ref> | '(' <
    condition> ')'
4  <fact-ref>      ::= ID '(' <arg> ( ',' <arg> )* ')' ?

```

A condition is a boolean expression over declared Facts, and the only kind of guard allowed in an IF branch. The operators NOT, AND, and OR bind in that order, from tightest to loosest, and parentheses override this precedence. The condition of the `food_served` plan, `(is_hungry OR is_drunk) AND NOT hates(host)`, uses all three operators and a parenthesised group. Table 5.3 lists the translation rules; the compiled context above shows the group preserved as `(is_hungry | is_drunk)`.

Regia	AgentSpeak
IF is_hungry:	Context clause : is_hungry
IF NOT is_drunk:	Context clause : not is_drunk
IF a AND b:	Context clause : a & b
IF (a OR b) AND c:	Context clause : (a b) & c

Table 5.3: Condition operators and their AgentSpeak translation.

5.3.5 Priority and Temper

```

1  <priority>      ::= 'PRIORITY' NUMBER
2  <temper>        ::= 'TEMPER' <temper-entry> ( ',' <temper-entry> )* <
    effects>?
3  <temper-entry> ::= ID '(' FLOAT ')'
4  <effects>       ::= 'EFFECTS' <temper-entry> ( ',' <temper-entry> )*

```

A WHEN block may carry further optional annotations, meant to affect plan selection when

several plans match the same Event. **PRIORITY** compiles to the plan annotation **priority** (**N**). It is meant to resolve conflicts between plans that share a trigger and context, letting one override the other. Jason, however, does not enforce it: as already discussed, **priority** is a reserved plan annotation in the Jason specification, defined for this purpose, but the default selection function ignores it (Section 2.3.6) [8]. Regia inherits this limitation, and Chapter 9 returns to it.

TEMPER integrates with the model of personality and mood that VEsNA gives each agent (Section 2.6.2) [19]. It annotates the plan with a weight for one or more dimensions of the agent’s mental state, and the selection function prefers the plans most compatible with the agent’s current values. **EFFECTS** declares how executing the plan shifts those dimensions.

The `insult_heard` plan (lines 33 through 38 of Listing 5.1) carries all three annotations: a sober, resentful Guest complains only if their sympathy for the host is already low, and complaining raises their aggressiveness. All three compile to plan annotations rather than context clauses, since VEsNA implements its model in the interpreter’s selection function, not in belief matching:

```

1 @pb__NobleSocializing__insult_heard__[priority(5),
2   temper([sympathy(-0.5)]), effects([aggressiveness(0.2)])]
3 +insult_heard : playbook_active(noblesocializing, _)
4   & not (is_drunk) & hates(host) <-
5   complain;
6   !signal_directors(noblesocializing, insult_thrown).
```

5.4 Plots

A Plot is a structured narrative scenario. It describes the phases a scenario passes through, the Roles agents play in it, and how an orchestrator, the Director, manages transitions and world-level events. Unlike a Playbook, a Plot is not attached to a single agent. Every active Plot instance spawns its own Director agent, which perceives events, tracks the current phase, and directs the agents bound to the Plot’s Roles. An individual agent does not know that it takes part in a Plot: it only receives and executes directives, so its Playbooks remain reusable across narratives. The Director’s logic compiles into one AgentSpeak file per Plot (Section 6.1.4), whose plans carry the label prefix `@dir__`.

```

1      <plot-def>      ::= 'PLOT' ID '.' <plot-header> <during-block>+

```

The running example declares two Plots, `HonorDuel` and `RoyalBanquet` (lines 56 through 106 of Listing 5.1). We focus on `RoyalBanquet`, the outer Plot, and return to `HonorDuel`, its Subplot, in Section 5.4.7.

5.4.1 Phases and Roles

```

1      <plot-header>    ::= (<phase-decl> | <role-decl>)+
2      <phase-decl>     ::= 'PHASE' ID 'INITIAL' '.' | 'PHASE' ID '.'
3      <role-decl>      ::= 'ROLE' ID '.'

```

The Plot header, which must precede every DURING block, declares the Phases and Roles of the scenario. A Phase is a named stage of the Plot, and exactly one Phase is active at any moment. `RoyalBanquet` declares three, `reception`, `feast`, and `duel_interruption`, and marks `reception` as `INITIAL`. The compiler rejects a Plot that does not declare exactly one `INITIAL` phase. The Director tracks the active Phase in a `current_phase(Name)` belief, which the compiled `RoyalBanquet` Director initialises to `reception`:

```

1  plot_name(royalbanquet).
2  current_phase(reception).
3
4  @boot__RoyalBanquet[atomic]
5  +!boot <-
6      .my_name(Me);
7      +plot_id(Me);
8      .print("[RoyalBanquet] Booting up director agent ", Me,
9              " in phase 'reception'...");
10     !on_enter(reception).

```

A Role is a named placeholder for the agents that participate in the Plot. It separates the script, which refers only to Role names such as `Host` and `Guest`, from the concrete agents bound to them at runtime, so that `RoyalBanquet` can run unmodified with a different host and guest list each time. The Director records these bindings in a `role_agent(Role, Agent)` registry, which the infrastructure plan `start_plot` populates when the Plot

begins:

```
1 +!start_plot(Bindings) <-  
2   for ( .member(map(Role, Agents), Bindings) ) {  
3     for ( .member(Agent, Agents) ) {  
4       +role_agent(Role, Agent);  
5     }  
6   };  
7   !boot.
```

Each Role compiles into its own file, one per pair of Plot and Role, which includes the Playbooks assignable to it (Section 5.3); its plans carry the label prefix @role__.

5.4.2 DURING Blocks

```
1 <during-block> ::= 'DURING' ID ':' <during-content>+  
2                | 'DURING' 'PLOT' ':' <during-content>+
```

A **DURING** block scopes behaviour to a single phase or to the whole Plot. In the running example, **DURING reception:** and **DURING feast:** contain behaviour active only in the corresponding phase, and the compiler adds a matching `current_phase(...)` guard to every plan generated from them. **DURING PLOT:** contains behaviour active in every phase, with no phase guard. **RoyalBanquet** uses it for the `assassin_spotted` handler, which must fire whether the banquet is in reception, in full swing, or interrupted by a duel:

```
1 @dir__RoyalBanquet__assassin_spotted__[priority(9)]  
2 +assassin_spotted : true <-  
3   lock_doors;  
4   !send_to_role(host, achieve, call_guards);  
5   !end_plot.
```

The context is `true`: with no phase guard, the handler fires in any phase.

A **DURING** block contains **WHEN** blocks (Section 5.4.5) and the lifecycle hooks **ON ENTER** and **ON EXIT** (Section 5.4.3). Hooks may appear only in a phase-specific block, since a Plot-

wide block has no single phase to enter or exit. `RoyalBanquet` uses `ON ENTER` in `reception` to start the music and assign `NobleSocializing` to every `Guest`, and `ON EXIT` in `feast` to remove that Playbook when the duel interrupts the banquet.

5.4.3 Transitions

```
1 <inline-transition-stmt> ::= 'TRANSITION' 'TO' ID '.'
```

A `TRANSITION TO` statement in a `WHEN` block moves the Plot from its current phase to a new one. In `RoyalBanquet`, the `food_served` Event moves the Plot from `reception` to `feast`. Handling an insult ends with a transition from `feast` to `duel_interruption` (Section 5.4.7). A `TRANSITION TO` must be the last statement of its block or branch. It is not allowed in `DURING PLOT`, which has no single current phase to leave. Nor is it allowed in `ON ENTER` or `ON EXIT`, since a phase cannot be left while it is being entered or exited.

A transition runs in a fixed order. The Director first evaluates any enclosing `IF` condition. It then runs the `ON EXIT` hook of the current phase, updates the `current_phase` belief, and runs the `ON ENTER` hook of the new phase. The infrastructure plan `switch_phase`, shared by every Plot, implements this order:

```
1 @switch_phase_atomic[atomic]
2 +!switch_phase(Target) : current_phase(Current) <-
3     !on_exit(Current);
4     -current_phase(Current);
5     +current_phase(Target);
6     !on_enter(Target).
```

Teardown for a phase, such as unassigning `NobleSocializing` from every `Guest`, therefore always runs before setup for the next. As a result, an agent is never left with two conflicting Playbooks active at once.

5.4.4 Plot Termination

```
1 <plot-end-stmt> ::= 'END' 'PLOT' '.'
```

END PLOT terminates the current Plot instance and destroys its Director agent. If the Plot belongs to a hierarchy of Plots (Section 5.4.7), it first notifies the Plots above and below it.

Termination also affects the agents the Plot managed. A Playbook is active on an agent only relative to the Director that assigned it (Section 5.3.1). A Director that ceased to exist would otherwise leave every Playbook it assigned permanently active. Every Role therefore carries a handler for the end of a Plot it is enrolled in, which withdraws every Playbook that Plot assigned:

```
1 +!plot_ended(PlotId)[source(PlotId)] <-  
2   -playbook_active(_, PlotId).
```

An agent enrolled in several Plots keeps the Playbooks its other Directors assigned, since each activation records the Director it came from.

5.4.5 Director WHEN Blocks

```
1 <plot-when-block> ::= 'WHEN' ID <priority>? ':' <  
    plot-when-body>  
2 <plot-when-subplot-ends-block> ::= 'WHEN' 'SUBPLOT' ID 'ENDS' <  
    priority>? ':' <plot-when-body>  
3 <plot-when-role-signals-block> ::= 'WHEN' 'ROLE' ID 'SIGNALS'  
    ID <priority>? ':' <plot-when-body>
```

A Director WHEN block defines a reactive plan, like a Playbook WHEN block: it fires when its trigger occurs, may carry PRIORITY, and may use the branches of Section 5.3.3. The running example uses all three forms of trigger:

- WHEN assassin_spotted PRIORITY 9: reacts to the named Event, whichever agent emitted it.

- **WHEN SUBPLOT HonorDuel ENDS:** reacts when the HonorDuel Subplot terminates (Section 5.4.7).
- **WHEN ROLE Guest SIGNALS insult_thrown:** reacts to the `insult_thrown` signal only when an agent bound to the **Guest** Role emitted it.

None of the three forms may carry **TEMPER**. VEsNA models the personality and mood of an individual agent (Section 2.6.2), and a Director is an orchestrator, not such an agent, so a Temper threshold has no state to evaluate against.

For the third form, the Director checks the `source` annotation of the incoming message against its `role_agent` registry, so that only a signal sent by an agent bound to **Guest** satisfies the context:

```

1 @dir__RoyalBanquet__guest_signals_insult_thrown__0
2 +insult_thrown[source(Sender)] : current_phase(feast)
3   & role_agent(guest, Sender) <-
4   stop_music;
5   !start_subplot("honorduel", honorduel, [map(challenger, guest)]);
6   !switch_phase(duel_interruption).

```

5.4.6 Director Statements

```

1 <imperative-stmt> ::= <assign-stmt> | <unassign-stmt>
2                   | <world-do-stmt> | <role-do-stmt>
3                   | <inline-transition-stmt>
4                   | <start-subplot-stmt> | <plot-end-stmt>

```

The Plot's Director acts on the world and on other agents, not on itself, and so uses a different set of statements from Playbooks. These statements appear in **ON ENTER** and **ON EXIT** hooks and in Director **WHEN** blocks. Table 5.4 lists each statement with an example from **RoyalBanquet**.

ASSIGN NobleSocializing TO Guest compiles to an achievement goal sent to every agent bound to **Guest**. The goal records the Playbook as active in the agent's belief base:

Statement	Example from RoyalBanquet
ASSIGN Pb TO Role.	ASSIGN NobleSocializing TO Guest.
UNASSIGN Pb FROM Role.	UNASSIGN NobleSocializing FROM Guest.
RoleName DO action.	Host DO call_guards.
WORLD DO action.	WORLD DO play_music.
TRANSITION TO phase.	TRANSITION TO feast.
START SUBPLOT Plot MAPPING ...	START SUBPLOT HonorDuel MAPPING Guest TO Challenger.
END PLOT.	(executed by the assassin_spotted handler)

Table 5.4: Director statements available inside **ON ENTER**, **ON EXIT**, and **WHEN** bodies, with examples from RoyalBanquet.

```

1 +!on_enter(reception) <-
2   play_music;
3   !send_to_role(guest, achieve, add_playbook(noblesocializing)).

```

`RoleName DO action.` also addresses a Role, but differs in kind. `ASSIGN` installs a standing behaviour, a Playbook, that stays active until `UNASSIGN` removes it. `RoleName DO` sends a one-off achievement goal, executed once and not retained. In the `assassin_spotted` handler, `Host DO call_guards` asks the Host to perform `call_guards` immediately. The alias of Section 5.2 resolves this name to `engine.security.summon_guards`. Both statements compile to the same infrastructure plan, generated once per Director file:

```

1 @dir__RoyalBanquet__send_to_role__o
2 +!send_to_role(Role, Performative, Content) <-
3   .findall(A, role_agent(Role, A), Agents);
4   .send(Agents, Performative, Content).

```

`send_to_role` collects every agent bound to the named Role from the `role_agent` registry, and sends one message to all of them. The two statements differ only in the Role and the message they pass. This is what lets a script address `Guest` or `Host` without knowing which agents those names resolve to.

`WORLD DO` lets the Plot itself perform declared Actions and the special actions of Table 5.2, for effects that belong to no single agent. Through `BELIEVE` and `FORGET`, it also lets the Director store information at the Plot level.

5.4.7 Subplots

```
1 <start-subplot-stmt> ::= 'START' 'SUBPLOT' ID ('MAPPING' <role-  
    mapping> (',' <role-mapping>)*)? '.'  
2 <role-mapping> ::= ID 'TO' ID
```

A Plot may start child Plots, or Subplots, with `START SUBPLOT`, delegating part of its story to a smaller, self-contained scenario. The `MAPPING` clause binds Roles of the parent to Roles of the child. When a Guest signals `insult_thrown`, `RoyalBanquet` starts `HonorDuel`, mapping its `Guest` Role onto the `Challenger` Role:

```
1 START SUBPLOT HonorDuel MAPPING Guest TO Challenger.
```

`HonorDuel` is a minimal Plot: one phase, `combat`, and one handler that ends the Plot when a `duel_resolved` Event occurs. Parent and child communicate in both directions. The parent reacts to the end of the child through `WHEN SUBPLOT HonorDuel ENDS:`, compiled as a handler for the `child_ended` achievement goal:

```
1 @dir__RoyalBanquet__subplot_honorduel_ended__@  
2 +!child_ended(honorduel, _) : current_phase(duel_interruption) <-  
3   .print("The duel is settled. The banquet is ruined.");  
4   !end_plot.
```

Conversely, a Subplot reacts to the end of its parent through the reserved event `parent_ended`. Every compiled Director, including that of `HonorDuel`, carries this handler, whether or not it has a parent:

```
1 +!parent_ended[source(Parent)] : parent_plot(Parent) <-  
2   -parent_plot(Parent);  
3   !end_plot.
```

The handler always terminates the child when its parent ends; an author cannot specify a

different reaction, such as letting an orphaned Subplot continue. Chapter [9](#) returns to this as future work.

Chapter 6

Implementation

This chapter describes the two artifacts that realise Regia in practice: a Python transpiler that turns `.regia` source into AgentSpeak, usable from the command line, and a browser-based visual editor that shows a narrative’s structure as an interactive graph. We concentrate on how a Regia program becomes a set of AgentSpeak agents, and on the decisions in that translation that the language did not dictate. Chapter 5 gives the mapping of each construct, which we restate only where necessary.

The two artifacts are not independent implementations. The editor’s backend imports the transpiler as a library and calls its `compile_source()` entry point, so a program compiled in the browser passes through the same phases as one compiled from a terminal (Section 6.2.2). Both are available in a public repository [59]: the transpiler in `language/src/regia/`, with one module per compilation phase, and the editor in `editor/`, split between a Python backend and a React frontend. File names in this chapter refer to that layout.

Section 6.1 follows the transpiler’s pipeline, from source text to generated AgentSpeak, tracing one Playbook of the running example through it. Section 6.2 follows the editor’s two data directions, from source to graph and from graph to source, and closes with its error handling and limitations.

6.1 Transpiler

6.1.1 Architecture and Pipeline

The transpiler turns a Regia program into one or more AgentSpeak agents. It is written in Python, using Lark [33] for parsing and Click [46] for its command-line interface. We chose Python for its flexibility and the speed of prototyping it allows: the language changed

repeatedly in response to our collaborators’ feedback (Chapter 4), and each change had to reach the compiler quickly. It is a multi-pass transpiler: rather than translating source into target code in a single traversal, it applies a fixed sequence of independent transformations, each consuming the representation the previous one produced.

Two properties follow. Each phase stays narrow in scope: the preprocessor never needs to know what a Playbook is, and the validator never needs to know what an AgentSpeak plan looks like. The pipeline is also a sequence of plain functions, which another program can call as a library. The Visual Editor does this, and compiles Regia through the transpiler rather than through logic of its own.

The pipeline runs six phases, listed in Table 6.1 together with the module implementing each and the representation it produces.

Phase	Module	Produces
0. Preprocess	<code>preprocessor.py</code>	<code>SourceAnnotations</code>
1. Parse	<code>parser.py</code>	<code>lark.Tree</code>
2. Build AST	<code>ast_builder.py</code>	<code>Program</code>
3. Annotate	<code>compiler.py</code> (internal)	<code>Program</code> , with doc comments attached
4. Validate	<code>validator.py</code>	<code>Program</code> , validated, or failure
5. Emit	<code>emitter.py</code>	<code>Dict[str, str]</code>

Table 6.1: The six transpiler phases, the module implementing each, and the representation it produces.

Preprocessing scans the raw source for doc comments and import paths, which must be captured before anything else. Parsing hands the unmodified source to Lark, which produces a concrete syntax tree from the grammar. AST Construction rebuilds that tree as a set of typed, Regia-specific dataclasses. Comment Attachment reconnects the extracted doc comments to the AST nodes they precede. Semantic Validation checks the rules the grammar cannot express. Emission produces the AgentSpeak source.

Four entry points expose these phases. `compile_source` and `compile_file` run the full sequence; `parse_source` and `parse_files` stop once the AST is built. The file-based variants first resolve the import graph, since one file may bring in several others, and merge the result into a single `Program`. `compile_source` receives a string rather than a path, and does not resolve imports. Each entry point returns a `CompileResult` carrying the generated files, the diagnostics, and the AST. Passing `emit=False` runs every phase except Emission, validating a program without generating AgentSpeak.

A single `ErrorReporter`, created at the entry point, runs alongside the phases rather than between them. Every phase up to Semantic Validation writes its diagnostics into it, and the pipeline stops after the first phase that reports an error. Emission runs only on a valid program, and reports nothing. Section 6.1.5 describes what the reporter produces. The following sections follow the pipeline in order, using excerpts from Listing 5.1.

6.1.2 From Source to AST

The first four phases turn the source text into an AST, a structure closer to the language’s semantics, on which validation and emission operate.

Two things must be handled before the parser runs. The first is doc comments. The grammar discards any line beginning with `#` during lexing, which is correct for an ordinary comment but not for a doc comment, whose `#@key: value` form (Section 5.1.1) carries metadata meant to be read. `preprocess()` recovers that text before the lexer reaches it, scanning the file line by line and recording each annotation’s key, value, and line, together with any `#-` continuation lines. The `NobleSocializing` Playbook carries one:

```
1 #@desc: Reactive behaviour for guests mingling and eating at the  
   banquet.  
2 PLAYBOOK NobleSocializing:
```

No `#-` line follows, so the annotation is closed when the `PLAYBOOK` line is reached, and stored apart from the code it describes.

The second is imports. An `IMPORT "path".` statement matches a grammar rule and survives lexing, but the compiler must know which other files a compilation involves before parsing, to process them together. `resolve_imports` walks the import graph from the entry file and returns the ordered list of files involved, entry file first. A file imported from two places is read once, so a diamond, two files importing a third, is accepted, while an import chain leading back to its start is reported as a cycle. Neither a cycle nor an unreadable file halts the walk, so an author with two broken imports sees both reported in one run. The running example declares no imports; had its vocabulary been moved into a separate file, as Section 5.1 suggests, resolution would return both files before either was parsed.

The grammar is compiled once, at import time, into an LALR table that every later call reuses. An LALR parser reads its input once, left to right, and decides each step from a single token of lookahead. Its grammar therefore cannot contain rules that need a wider view of the program, and any ambiguity must be resolved by the shape of the rules themselves; Regia’s condition rules rely on this (Section 6.1.4). Lark is also configured to attach a line and column to every node it builds, not only to tokens. This is the only origin of source positions in the system, and every diagnostic of Section 6.1.5 traces back to it. Malformed input is not recovered from: Lark raises an exception, which the entry point translates into a diagnostic an author can act on.

AST Construction turns the parse tree into the typed structure every later phase reads. The parse tree mirrors the grammar rather than the language, with a node for every rule that fired, including infrastructural ones. Later phases need a tree whose nodes are Regia’s constructs. The build runs bottom-up: a builder method is called on each node, from the leaves to the root, with children already transformed. Every node produced is a dataclass carrying its information and a source location derived from the parser’s positions. Figure 6.1 shows the resulting hierarchy.

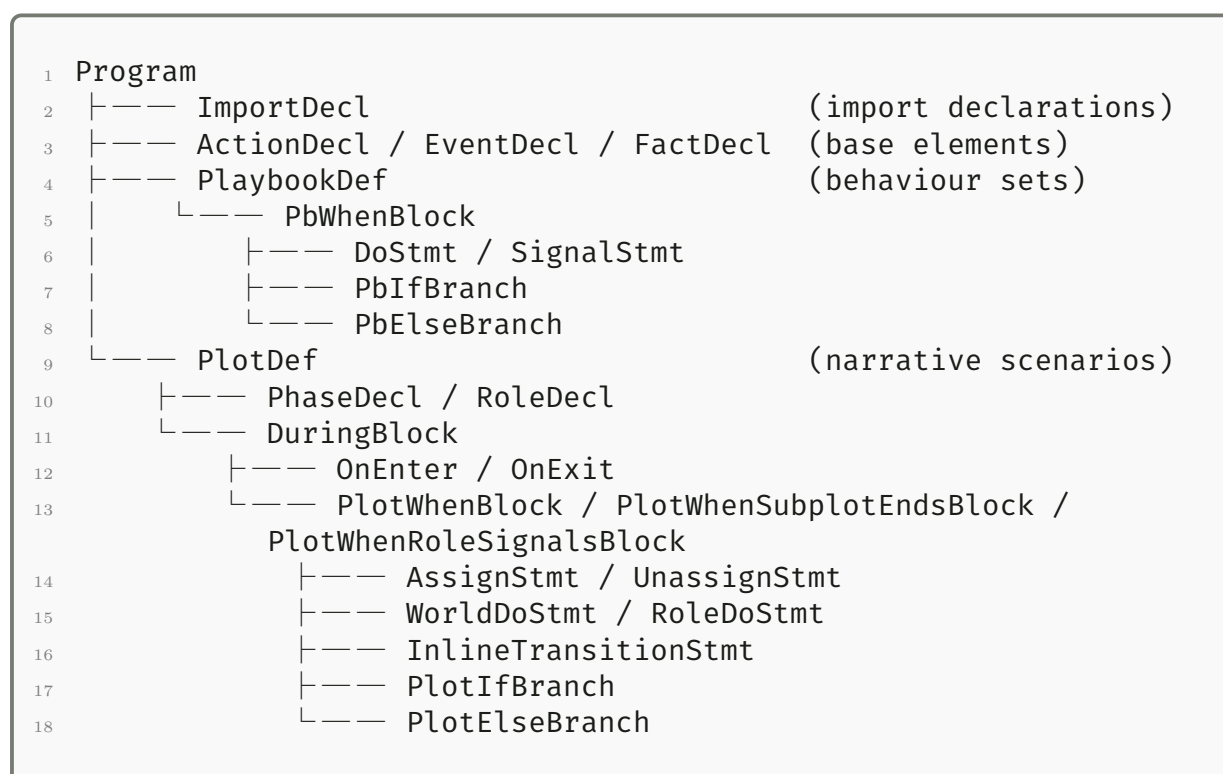


Figure 6.1: The AST node hierarchy produced by AST Construction, and the structure every subsequent phase operates on.

NobleSocializing occupies the **PlaybookDef** branch of this hierarchy. It builds one **PlaybookDef** holding three **PbWhenBlock** nodes, one per **WHEN** block. The **insult_heard** block splits into a **PbIfBranch** and a **PbElseBranch**, each holding **DoStmt** nodes, and the **SIGNAL** statement in the first becomes a **SignalStmt**. The **food_served** block adds a **DoStmt** ahead of its two branches.

Comment Attachment then merges the doc comments with the AST, now that every construct has a node with a position. It sorts the candidate declarations by line and attaches each annotation, in source order, to the first declaration after it. The **desc** annotation of the running example therefore attaches to the **PlaybookDef**. An annotation with no

declaration after it is attached to the program as a whole, as a file-level description, rather than discarded.

6.1.3 Semantic Validation

The grammar guarantees that a program is well formed, not that it means anything: that a referenced Event is declared, that a Plot has a starting Phase, or that a **TRANSITION TO** sits where it can execute. Chapter 5 states rules of this kind as properties of the language; Semantic Validation checks them on a given program, reading the AST and writing diagnostics into the reporter.

Two scopes govern every check. One flat table holds four global namespaces, for Actions, Events, Facts, and Playbooks, each visible from anywhere in the program. Each Plot has a scope of its own, holding its Roles and Phases. A Role name means something only in the Plot that declares it, so two Plots may each declare a Role called **Guest** without conflict.

Table 6.2 lists the rules enforced, the place in Chapter 5 where each is stated, and whether a violation is reported as an error or a warning.

The checks run in three steps. Every declaration is collected first, so that a reference resolves wherever the name appears; a Plot may start a Subplot defined below it. Each Playbook and Plot is then walked once, with structural and reference checks interleaved. Unused-name warnings run last, since a name cannot be judged unreferenced until every walk has finished.

NobleSocializing exercises the reference rules. Each of its **WHEN** blocks names an Event that must be in the global Event namespace. Each **IF** condition resolves its Facts there too, with matching arity: **is_hungry** and **is_drunk** take no argument, while **hates(host)** supplies the one argument **FACT hates(person).** declares. Each **DO** resolves its Action the same way. **DO PRINT("A guest finishes their meal.")** skips the existence check, since **PRINT** is built into the language, but its number of arguments is still checked.

The unused-name check accounts for Action aliases. For **ACTION engine.security.summon_guards AS call_guards.**, the validator registers both names as counterparts, and a reference to **call_guards** marks both as used. Otherwise, an author using the alias as intended, declaring the qualified name once and writing only the short form, would see the qualified name reported as unused on every compile.

6.1.4 AgentSpeak Emission

Emission, the final phase, translates the validated AST into AgentSpeak. **Emitter.emit(program)** returns a **Dict[str, str]** mapping output filenames to AgentSpeak source, and

Category	Rule	Stated in
Declarations	A name is declared once in its namespace	Section 5.2
	A Plot declares exactly one INITIAL Phase	Section 5.4.1
Structure	A DURING block names a Phase of its own Plot	Section 5.4.2
	A DURING block holds at most one ON ENTER and one ON EXIT	Section 5.4.2
	TRANSITION TO and END PLOT do not appear inside ON ENTER or ON EXIT	Section 5.4.3
	TRANSITION TO does not appear inside DURING PLOT	Section 5.4.3
Position	TRANSITION TO and END PLOT are last in their statement list	Section 5.4.3
References	An Event, Action, or Fact referenced is declared, with matching arity	Section 5.2
	A Playbook named in ASSIGN or UNASSIGN is defined	Section 5.3
	A Role named anywhere is declared by the Plot naming it	Section 5.4.1
	A Phase targeted by TRANSITION TO is declared by its Plot	Section 5.4.3
	A Plot named by START SUBPLOT or WHEN SUBPLOT ... ENDS is defined, and every Role in a MAPPING clause exists on the side that declares it	Section 5.4.7
Warnings	A START SUBPLOT omits MAPPING for a Plot that declares Roles	Section 5.4.7
	A declared name is never referenced	Section 5.2

Table 6.2: The semantic rules the Validator enforces, the section of Chapter 5 stating each, and the severity reported on violation.

never touches the filesystem; the CLI writes the result to disk (Section 6.1.6).

6.1.4.1 Generated Files

Emission produces three kinds of file, listed in Table 6.3.

Filename	One per	Contents
<code>playbook_<n>.asl</code>	Playbook	Gated WHEN plans
<code>director_<plot>.asl</code>	Plot	Phase state machine, lifecycle infrastructure, Director WHEN plans
<code>role_<plot>_<role>.asl</code>	Plot × Role	Playbook includes, activation handlers, one-off command handlers

Table 6.3: The three kinds of file Emission produces, and what each contains.

The running example produces six files: `playbook_noblesocializing.asl`, one Director file for each of its two Plots, and one Role file for each pair of Plot and Role, `role_royalbanquet_host.asl`, `role_royalbanquet_guest.asl`, and `role_honorduel_challenger.asl`. A Role file does not contain a Playbook’s plans, but includes them through Jason’s file-inclusion directive:

```
1 { include("playbook_noblesocializing.asl") }
```

Each Playbook’s compiled form therefore lives in one place, and a correction to `NobleSocializing` reaches every Role file that includes it on the next compile. A Director file includes nothing, since everything it contains belongs to its own Plot.

These files are agent source and nothing else. A running system also needs a multi-agent configuration binding each Plot’s Roles to running agents, an implementation behind every declared Action, and the engine-side handling that gives each Action an effect in the game world. Section 7.1.1 describes a system that provides them.

6.1.4.2 Which Playbooks a Role File Includes

An agent runs the program it was launched with, so every Playbook it may ever receive must be included in its Role file before it starts. A Director does not add plans at runtime; it only activates plans already present, through the `playbook_active` guard of Section 5.3.1.

Subplots make this set larger than the Playbooks a Plot assigns to the Role. In the running example, `RoyalBanquet` maps its `Guest` Role onto the `Challenger` Role of `HonorDuel`. An agent playing `Guest` may therefore also play `Challenger` while the duel runs, and receive whatever `HonorDuel` assigns to that Role. `role_royalbanquet_guest.asl` must include those Playbooks alongside `NobleSocializing`. `role_honorduel_challenger.asl`, whose Role is mapped nowhere further, includes only what `HonorDuel` assigns.

In general, the emitter treats every `MAPPING` clause as an edge from one pair of Plot and Role to another, and a Role file includes the Playbooks of every pair reachable from its own. The search follows chains of edges, so Subplots nested at any depth are handled like the single level of the running example.

The same structure explains why Subplots expand more than any other construct. Each one emits a Director file and a Role file of its own, as much output as a second, minimal program; Section 7.2.2 measures the resulting expansion.

6.1.4.3 Writing Plans

Every plan is written by one shared routine, so that annotations, trigger, context, and body are formatted the same way throughout. A plan with default priority and no Temper carries no annotations, and a missing context or empty body is written as `true`, so every generated plan is syntactically complete.

Most statements translate through the mappings of Chapter 5: the special actions of Table 5.2 and the Director statements of Table 5.4. Two need more than a substitution. `START SUBPLOT` turns each of its mappings into a `map(target_role, source_role)` entry. A one-off Role directive dispatches a goal named after its Action, and the dots of a qualified name such as `engine.security.summon_guards` cannot appear in a goal name, so the emitter replaces them with underscores.

6.1.4.4 Conditions

The grammar gives `NOT`, `AND`, and `OR` their precedence through the shape of its rules, with no precedence table: an `AND` can contain an `OR` only inside explicit parentheses, so the AST always records the grouping the author wrote (Section 5.3.4). Jason, however, like Prolog, binds `&` tighter than `|`, and writing the operators out in sequence would lose that grouping. The emitter therefore wraps a disjunction in parentheses whenever it is an operand of a conjunction, and always wraps the operand of a negation, which is safe since `not` applies to the whole expression it encloses.

The condition of the `food_served` block shows both rules:

```
1 IF (is_hungry OR is_drunk) AND NOT hates(host):
```

```
1 (is_hungry | is_drunk) & not (hates(host))
```

6.1.5 Diagnostics

Every phase up to Semantic Validation writes into one **ErrorReporter**, created in `compile_source`. The reporter is not a phase: it consumes and produces no representation. It is the one component through which the compiler addresses the author.

Its output matters more here than in a general-purpose compiler. Regia’s audience does not write code (Section 4.1), and its syntax is grounded in what that audience can read without training (Section 4.2). A compiler serving this audience must report failures on the same terms. A diagnostic that names the phase that detected a problem, or the grammar terminal that failed to match, would return the author to the position Regia exists to move them out of.

Every diagnostic therefore takes one shape, whichever phase raised it:

```
1 @dataclass
2 class CompilerMessage:
3     severity:    Severity
4     line:        int
5     column:      int
6     length:      int
7     message:     str
8     hint:        str = ""
9     source_line: str = ""
10    filename:     str = ""
```

The fields hold what an author needs to act: where the problem is, what it is, how to correct it, and the offending line with a caret under the column. Because every diagnostic has this shape, `CompileResult.messages` is one flat list, and two renderers, the terminal and the Visual Editor (Section 6.2.2), use the same record unchanged.

Suppose the running example’s `ACTION lock_doors.` declaration were mistyped as `ACTION`

`lock_door..` The `assassin_spotted` handler still calls `WORLD DO lock_doors..`, so the single typo produces two diagnostics, from two checks of Section 6.1.3:

```
=====
WARNING  royal_banquet.regia:10, col 0
Declared action 'lock_door' is never used.
  ACTION lock_door.
    ^
=====
=====
ERROR    royal_banquet.regia:79, col 12
Undeclared action: 'lock_doors'.
Hint: Add 'ACTION lock_doors.' at the top of the file.
      WORLD DO lock_doors.
      ^
=====
```

The hint states the correction to make, not the rule that was broken. Messages are sorted by file and line, so an author reads them in source order rather than in the order the compiler found them.

6.1.5.1 Syntax Errors

Parsing is the one phase that does not report through `ErrorReporter` directly. Lark raises its own exceptions, and one function translates them into the shape above. The accessibility requirement is hardest to meet here, since a parse failure carries no semantic information: the parser knows only which token it found and which terminals it expected.

Terminal names are written for a parser, not a reader. A small table translates the ones an author is likely to meet: `ID` is reported as a name, with examples, `NUMBER` as an integer, and `STRING` as a text string. The table also covers the keywords of Table 5.2. A terminal absent from the table is reported by its quoted name.

Relabelling is not enough in every case. An `IMPORT` written after the first item violates the ordering constraint of Section 5.1. The generic path would report an unexpected token and a list of expected ones, which is accurate but unhelpful: the problem is where the statement is, not what it is. The translation function intercepts this case and reports that an `IMPORT` was found in an unexpected place, stating the rule that imports belong at the top of a file.

6.1.6 Invoking the Compiler

`cli.py` exposes the entry points of Section 6.1.1 as Click commands. `compile` runs the full pipeline, prints the diagnostics and a pass or fail summary, and writes the generated files to an output directory. `check` runs the same pipeline with `emit=False`. A `--quiet` flag suppresses warnings and progress output, and a hidden `dump-ast` command prints the AST, for debugging the compiler itself. The Visual Editor calls the same `compile_source` function in place of a terminal.

6.1.7 Testing

An automated suite of 217 tests, run with `pytest` [50], covers the transpiler. The tests were written alongside it and updated as the language changed. Each test compiles a Regia source through the full pipeline and compares the result with an expected output. For a valid program, the content of every generated AgentSpeak file must match an expected string. For an invalid program, the diagnostics the compiler reports must match the expected errors, which also tests the reporting of Section 6.1.5. The suite shows that the transpiler produces the expected output for the programs tested, not that its translation is correct in general.

6.2 Visual Editor

The second artifact is a visual editor for Regia, built on top of the transpiler. It answers R4 (Section 4.5) for a narrative’s structure, its Phases and the transitions between them, which is the part of a Regia program the designers’ own sketches described. The editor was added after the transpiler, in response to the designers’ feedback, and is less complete; Section 6.2.4 sets out what it does not yet support.

The tool is a single-page web application, split across two processes: a Python backend wrapping the transpiler, and a TypeScript frontend running in the browser. We chose a web application for its portability: it runs in any modern browser, on any operating system, and a designer can use it without installing a toolchain. Existing components also provided most of its interface, Monaco for code editing and React Flow for graphs, while the Python backend reuses the transpiler unchanged. Table 6.4 lists the technology behind each part. The architecture is otherwise conventional, so we describe the tool through its two data directions: from source to graph (Section 6.2.2) and from graph to source (Section 6.2.3).

Component	Technology
Backend service	Python, FastAPI [51]
Frontend application	TypeScript, React [41]
Code panel	Monaco Editor [42]
Graph panel	React Flow [62], with Dagre [12] for layout

Table 6.4: The technology behind each component of the Visual Editor. The transpiler it wraps is described in Section 6.1.

6.2.1 Overview and Use

The interface is split into two panels, shown in Figure 6.2. On the left is a code editor built on Monaco, the editor of Visual Studio Code, configured with a Regia language definition. On the right is a graph canvas built on React Flow, a React library for node-and-edge diagrams, with node positions computed by Dagre. Neither panel holds its own copy of the program: both read from one shared store, holding the current source text, the AST last compiled from it, and the resulting diagnostics. The two panels therefore stay in sync as the author types, with no manual refresh.

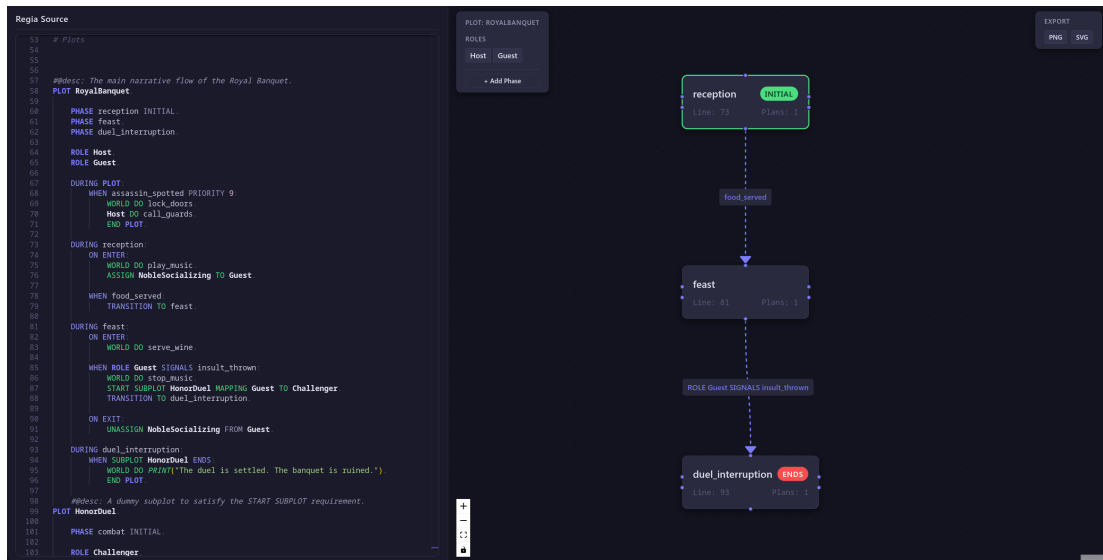


Figure 6.2: The Visual Editor's two-panel layout: a Monaco code editor on the left, and a React Flow graph canvas on the right showing the Phases of a Plot and the transitions between them.

Loading `royal_banquet.regia` into the code panel populates the canvas. A node appears for each of the three Phases, with `reception` marked as the `INITIAL` Phase, and an edge connects each pair of Phases linked by a `TRANSITION TO`, labelled with its trigger.

The graph stays live as the source changes. Renaming `duel_interruption` to `duel_aftermath` updates the node a moment after the author stops typing. A mistake produces two signals:

the offending line is highlighted in the code panel, and the canvas is dimmed to show that it may no longer match the source. Correcting the error clears both.

The canvas also supports two edits. Adding a Phase prompts for a name, then adds a node to the graph and a **PHASE** declaration to the source. Adding a transition is more involved. Suppose an author wants a Phase reached once the duel is over. They add a Phase named **aftermath** and drag a connection from **duel_interruption** to it. Before the edge is drawn, a prompt asks which Event triggers the transition. Typing **guests_leave** completes the interaction: the edge appears, and the code panel gains an **EVENT** **guests_leave.** declaration near the top of the file and a **WHEN** **guests_leave: TRANSITION TO aftermath** . block inside **DURING** **duel_interruption:**, where an author would have written it by hand. Section 6.2.4 explains why these are the only two edits supported.

6.2.2 Source to Graph

Figure 6.3 traces a single edit from keystroke to redrawn graph.

The backend is a small FastAPI application with one endpoint, **POST /parse**, which receives the source text and calls **compile_source** with **emit=False**, reusing the entry point of Section 6.1.1 unmodified. Emission is skipped, since the editor needs a validated program to draw, not AgentSpeak. No parsing or validation logic is duplicated in the editor.

The response carries the AST of Figure 6.1, serialised to JSON. Every node carries the name of its class alongside its fields, so the frontend can dispatch on node kind rather than infer it from shape. The frontend never calls the network directly: every request goes through a small **Transport** interface, currently backed by one HTTP implementation, so that other sources of compilation can be added as further implementations. A single function, **convertAstToGraph**, turns the returned program into React Flow's nodes and edges. It maps only the first **PlotDef** in the program.

Every **PhaseDecl** in that Plot becomes a node, recording the line the Phase is declared on, how many **WHEN** blocks its **DURING** block contains, whether that block has **ON ENTER** or **ON EXIT** hooks, and whether the Phase is terminal. A Phase is terminal when a statement reachable from any of its **WHEN** blocks, in the prefix statements or in any branch, ends the Plot. In the running example, **duel_interruption** is terminal, since its **WHEN** **SUBPLOT HonorDuel ENDS** handler reaches **END PLOT**. Terminal Phases are drawn distinctly, so an author sees which Phases end the narrative without reading their handlers.

Edges come from every **TRANSITION TO** statement in a phase-specific **DURING** block; Plot-wide blocks have no single source Phase and are skipped. An edge's label depends on the form of the Director **WHEN** that produced it (Section 5.4.5): the Event name for a plain trigger, the Role and signal for **WHEN** **ROLE ... SIGNALS**, and the Subplot name for **WHEN**

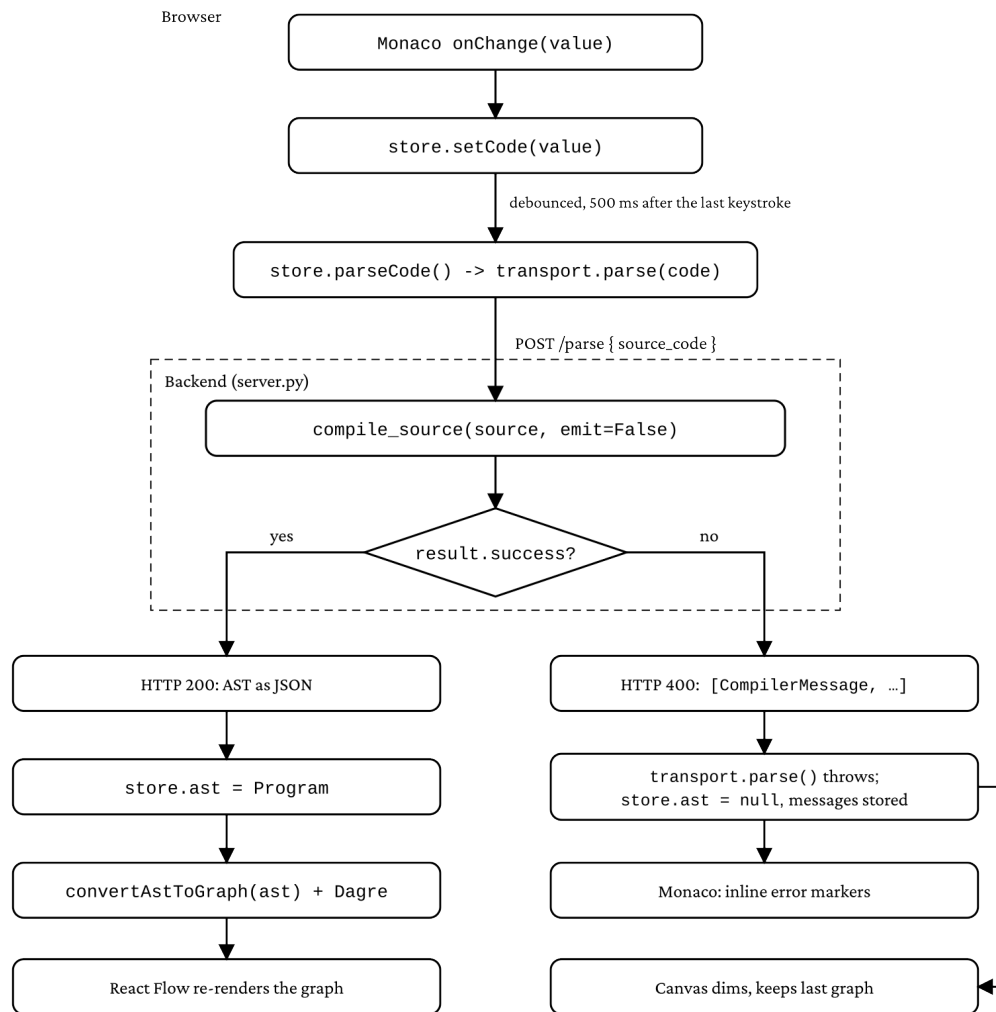


Figure 6.3: The path from a keystroke in the code panel to a redrawn graph, passing through the transpiler itself rather than through any parsing logic held by the editor.

`SUBPLOT ... ENDS`. The graph therefore uses the language’s own vocabulary rather than a generic state-machine one. When two Phases are connected by more than one transition, the first edge is drawn straight and further edges curve to alternating sides, so that parallel transitions remain legible.

Nodes and edges carry no positions: Dagre computes the layout in a separate pass, so the author never arranges the graph by hand and can focus on the logic it represents.

6.2.3 Graph to Source

An edit on the canvas takes the opposite path, and does not reach the backend. The frontend writes the corresponding Regia text directly, treating the source as a list of lines and using positions from the current AST. Adding a transition inserts an `EVENT` declaration before the first `PLOT`, if the Event is not yet declared, and a `WHEN` block at the end of the source Phase’s `DURING` block. Adding a Phase inserts a `PHASE` declaration after the last one, and a matching `DURING` block with a starter `ON ENTER` hook. Each operation offsets its second insertion by the lines the first added, and normalises the typed name to `snake_case`, matching the identifier conventions of the language.

Writing source on the frontend is an interim measure. The `Transport` interface anticipates a second method, sending a modified AST to the server and receiving regenerated source; the module that would build that AST from the canvas exists as a documented stub. Under that design, the server would remain the single source of truth for parsing and emission, and generated source would be valid by construction. The current approach offers no such guarantee, which is why the canvas supports two operations rather than an open-ended set.

6.2.4 Error Handling and Limitations

6.2.4.0.1 Error handling. The editor recompiles the source shortly after each edit, so an author sees problems while typing rather than on request. A program that fails to compile takes the error branch of Figure 6.3: the backend returns diagnostics instead of an AST, and the code panel shows each one as an inline marker at its line and column, with its message and hint. These are the same `CompilerMessage` records the CLI prints (Section 6.1.5), produced once and rendered by two independent front ends, so an author reads the same domain-level wording in both. The canvas dims rather than clearing: an author whose program does not currently compile still sees the last structure that did, and the dimming signals that it may no longer match the source. The next successful compile restores it.

6.2.4.0.2 Limitations by design. The canvas draws a narrative’s structure and nothing else: its Phases, the transitions between them and their triggers, and which Phases start and end the Plot. Playbooks, the vocabulary, Director statements other than transitions, and the reactive logic inside a Playbook have no visual form, and an author edits them in the code panel. This boundary follows from R4, which asked for a way to see a narrative as our collaborators sketched one, not to visualise every construct (Section 4.5).

6.2.4.0.3 Unfinished work. Two further limitations are not intended. First, only the Plot declared first in a program is drawn: the others, and the Subplot hierarchy connecting them, can be inspected only in the code panel. Second, the canvas supports two edits, adding a Phase and adding a transition; renaming, deleting, or rewiring a Phase requires editing the source, since text written by the frontend is not guaranteed to form a valid program (Section 6.2.3). The reverse path that would lift this restriction, sending a modified AST to the server and receiving regenerated source, is designed but not implemented. Section 9.2 returns to both.

Chapter 7

Experiments

This chapter puts Regia to practical use in two ways. Section 7.1 describes *Everything For Science!*, a complete game built with Regia, and what its development revealed about using the language. Section 7.2 measures how the transpiler’s compilation time and output size scale as a project grows.

7.1 “Everything For Science!” Test Case

This section shows Regia used as a whole, to build a working game. *Everything For Science!* is a short game running in Godot through VEsNA, in which a Player must retrieve a set of items while interacting with a small cast of agents. It exercises several constructs at once: three Roles with independent Playbooks, a main Plot that branches on the Player’s choices, and a Subplot that starts, runs, and ends within the main narrative. It therefore shows that Regia can represent a complete, branching narrative, beyond the individual examples of Chapter 5.

Section 7.1.1 describes how a compiled Regia program is deployed and connected to Godot through VEsNA. Section 7.1.2 describes the game’s plot, as a narrative and through the Regia constructs that realise it. Section 7.1.3 reports the compilation metrics and the observations that came from authoring the game, several of which return in Chapter 9.

7.1.1 Infrastructure

This subsection describes how *Everything For Science!* uses VEsNA (Section 2.6.2): how a compiled Regia program is deployed, how a running agent communicates with the engine,

and what a developer must still build by hand. The game’s Regia source, internal actions, and engine-side handling are available in `science_game/` of the repository [59].

7.1.1.1 Deployment

A running Regia program is deployed as a JaCaMo multi-agent system, configured through a `.jcm` file. *Everything For Science!* declares four agents: one for each of the three Roles, `scientist`, `cop`, and `dog`, and one for the Director of the main Plot. Each agent is an instance of `vesna.VesnaAgent`, connected over its own WebSocket to a distinct port of the same Godot process. The Director has a port as well, although it has no visible body, since it may still relay environment information or perform internal actions. Its entry also carries a startup goal, `start_plot`, which binds the Plot’s Roles to the three running agents and begins execution; this binding is supplied at deployment, and appears nowhere in the compiled Regia output. Subplot Directors are not declared: when a `START SUBPLOT` statement executes, the parent Director creates the Subplot’s Director at runtime through Jason’s `.create_agent` internal action, and sends it its Role bindings.

7.1.1.2 Message Protocol

Once connected, an agent and its body in Godot exchange JSON messages in both directions. Table 7.1 lists every message type used in *Everything For Science!*. Messages from Godot to the mind arrive through `VesnaAgent`’s message handler and are converted into Jason percepts; messages from the mind to Godot are sent by internal actions, one per declared Regia Action, discussed next.

7.1.1.3 Internal Actions

Every Action declared in a Regia program must correspond to a Java internal action, implemented once, that packages its arguments as JSON and sends them to the body. Table 7.2 lists the fourteen Actions available to agents in *Everything For Science!*, grouped by what they affect in the game. Their narrative use is described where each first appears, in Section 7.1.2.

Listing 7.1 shows one such internal action in full, `add_dialogue_option`. It reads its four arguments, builds the corresponding JSON payload, and sends it to the body through `perform`; every other internal action in *Everything For Science!* follows the same pattern.

Direction	Message Type	Effect
Godot → Mind	regia_event	Becomes a flat, parameterless percept; this is the channel a Regia EVENT arrives through.
Godot → Mind	signal	Becomes a percept of the form signal_TYPE(status, reason) , carrying more detail than a plain event.
Godot → Mind	navigation	Becomes the percept reached_WAYPOINT once a movement action completes.
Mind → Godot	(any Action)	Carries a JSON payload specific to the Action performed; handled on the Godot side by VesnaManager .

Table 7.1: Message types exchanged between a Jason agent and its Godot body, and their effect on the receiving side.

```

1 public class add_dialogue_option extends DefaultInternalAction {
2
3     @Override
4     public Object execute(TransitionSystem ts, Unifier un, Term[] args)
5         throws Exception {
6         if (args.length != 4) {
7             throw new JasonException("vesna.via.add_dialogue_option
8                 requires exactly 4 arguments (id, text, event, close_ui
9                 ).");
10        }
11
12        VesnaAgent agent = (VesnaAgent) ts.getAg();
13
14        String id = args[0].toString().replace("\\", "");
15        String text = args[1].toString().replace("\\", "");
16        String event = args[2].toString().replace("\\", "");
17        boolean closeUi = args[3].toString().replace("\\", "").equals("
18            true");
19
20        JSONObject opt = new JSONObject();
21        opt.put("id", id);
22        opt.put("text", text);
23        opt.put("event", event);
24        opt.put("close_on_select", closeUi);
25
26        JSONArray optionsArray = new JSONArray();
27        optionsArray.put(opt);

```

Category	Action	Effect
Presence and Movement	<code>set_visible(visible)</code>	Toggles whether an agent is visible and interactable.
	<code>move_to(target)</code>	Moves an agent to a specified location.
	<code>start_wandering(radius)</code>	Begins erratic movement within a radius, simulating idle behaviour.
	<code>stop_wandering()</code>	Ends wandering movement.
Environment and Inventory	<code>spawn_item(item, waypoint)</code>	Makes an item appear at a location.
	<code>despawn_item(item)</code>	Removes an item from the environment.
	<code>add_item(item)</code>	Adds an item to the Player's inventory.
	<code>remove_item(item)</code>	Removes an item from the Player's inventory.
Dialogue	<code>utter(text)</code>	Shows a transient text bubble over an agent, without opening a dialogue.
	<code>set_dialogue_text(text)</code>	Sets the text shown in the main dialogue box.
	<code>add_dialogue_option(id, text, event, close_ui)</code>	Adds a choice the Player can select, optionally triggering an event and closing the dialogue.
	<code>remove_dialogue_option(id)</code>	Removes a single dialogue choice.
	<code>clear_dialogue_options()</code>	Removes every dialogue choice at once.
Control	<code>trigger_game_over(reason)</code>	Ends the game for the given reason.

Table 7.2: Internal actions available to agents in *Everything For Science!*, grouped by what they affect in the game.

```

24         JSONObject data = new JSONObject();
25         data.put("options", optionsArray);
26
27         JSONObject message = new JSONObject();
28         message.put("type", "add_dialogue_options");
29         message.put("data", data);
30
31         agent.perform(message.toString());
32         return true;
33     }
34 }
35

```

Listing 7.1: The `add_dialogue_option` internal action.

Regia and its transpiler are responsible for none of this infrastructure. The compiler produces `.asl` files and stops there; the `.jcm` configuration, the internal actions of Table 7.2, and the Godot-side message handling were written by hand, once, for this game.

7.1.2 Game Plot

A Scientist approaches the Player with a proposal. They ask to speak somewhere more private, and the Player follows them to their lab. There, the Scientist begins asking for items: first a flower, then a bone, then a corpse. Each request is framed as necessary for an unspecified experiment. A Cop patrols the area throughout. The Player can meet the Cop at any point and denounce the Scientist’s request; doing so affects the Cop’s mood, which we discuss at the end of this subsection. A Dog also wanders nearby. It plays little part in the story until the Player needs its help. At any point, the Player may instead drop something into the Scientist’s cauldron, which explodes and ends the game early. Figure 7.1 maps the Phases, branches, and endings described below.

7.1.2.1 Flower

The game begins in the Flower Phase, the Plot’s initial Phase. On entering it, the Scientist, Cop, and Dog each receive their starting Playbook. The Scientist spawns a flower in the garden, using the `spawn_item` Action (Table 7.2). The Scientist first asks the Player to follow them to the lab. Only once the Scientist signals that they have arrived does the Director swap their Playbook, from an introductory one to one that asks for the flower. The Player can meet the Cop at any point and denounce the Scientist’s request. The Dog only greets the Player in this Phase.

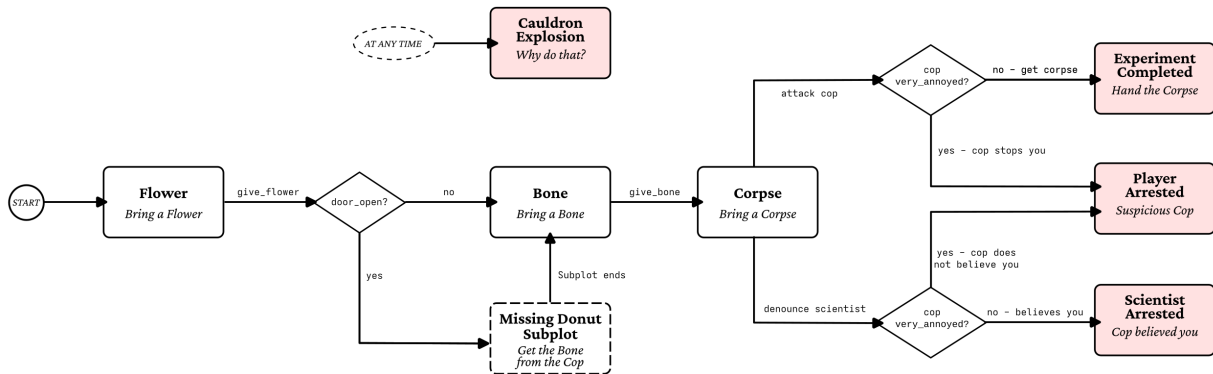


Figure 7.1: Phase and branch structure of *Everything For Science!*, from the Flower Phase to its four possible endings, tinted red.

Retrieving the flower requires the Player to open a garden door. The game tracks whether the door was left open as a persistent Fact, `door_open`. A Plot-wide handler sets this Fact, rather than a Phase-local one, so the game remembers the door’s state no matter when it was opened. Handing the flower to the Scientist branches the story for the first time, on this Fact:

```

1 WHEN give_flower:
2     IF door_open:
3         UNASSIGN CopFlowerPlaybook FROM Cop.
4         UNASSIGN DogMainPlaybook FROM Dog.
5         START SUBPLOT MissingDonut MAPPING Cop TO Cop, Dog TO Dog.
6         TRANSITION TO bone.
7     ELSE:
8         ASSIGN CopBonePlaybook TO Cop.
9         Dog DO BELIEVE(has_bone).
10        TRANSITION TO bone.

```

Listing 7.2: Branching on the garden door in the Flower Phase.

Leaving the door closed sends the Plot directly into the Bone Phase. The Dog already holds the bone the Scientist will ask for next. Leaving the door open instead starts a Subplot, described below.

7.1.2.2 Bone

If the door was left closed, the Bone Phase proceeds directly. The Dog, still holding the bone from the previous Phase, gives it up as soon as the Player asks.

If the door was left open, the Cop and Dog instead spend the Bone Phase inside **MissingDonut**, a Subplot. It maps both agents to new Roles for its duration. Its premise follows directly from the Player's earlier choice: the Dog used the open door to steal a donut from the Cop, and left the bone behind. The Cop refuses to hand over the bone without persuasion. The Player has two options: press the Cop directly, or first get the donut from the Dog and trade it for the bone.

Once the Subplot ends, by either path, control returns to the main Plot. The main Plot reacts to the Subplot's conclusion as a whole, rather than to any specific action taken inside it:

```
1 WHEN SUBPLOT MissingDonut ENDS:  
2     Scientist DO BELIEVE(player_has_bone).  
3     ASSIGN CopBonePlaybook TO Cop.  
4     ASSIGN DogMainPlaybook TO Dog.
```

Listing 7.3: The main Plot resuming after the Subplot ends.

Whichever path the Player took, the Bone Phase ends the same way as the Flower Phase before it. The Scientist accepts the bone, thanks the Player, and asks for a fresh corpse.

7.1.2.3 Corpse

The Corpse Phase offers the Player two ways to satisfy the Scientist's last request. This is also where the Cop's mood finally matters, a mood tracked since the Flower Phase as the Facts **cop_annoied** and **cop_very_annoied**. The Player can denounce the Scientist to the Cop one last time, or attack the Cop directly. Both outcomes depend on whether the Cop has reached its most annoyed state.

```
1 WHEN attack_cop:  
2     IF cop_very_annoied:  
3         DO set_dialogue_text("I saw you reaching! You are under arrest!"  
4         ").  
4         SIGNAL player_arrested.
```

```

5  ELSE:
6      DO utter("What are you doing?! *Dies*").
7      DO set_visible("false").
8      DO add_item("corpse").
9      SIGNAL cop_killed.

```

Listing 7.4: The Cop’s mood deciding the outcome of an attack.

As Listing 7.4 shows, a very annoyed Cop notices the attack coming and arrests the Player, ending the game. A less annoyed Cop is caught off guard instead. The Player gets the corpse, and a way to finish the experiment.

Denouncing the Scientist one last time follows the same logic, in reverse. A very annoyed Cop no longer believes the Player, and arrests them for the accusation itself. A less annoyed Cop believes them, and arrests the Scientist instead. Both endings use `trigger_game_over` (Table 7.2). Handing over the corpse, however it was obtained, completes the experiment and satisfies the Scientist. The game therefore has four endings: the experiment completed, the cauldron exploding, the Scientist arrested, and the Player arrested, the last reached by either attacking or denouncing a very annoyed Cop.

Two mechanisms appear in all three Phases. The first is the Cop’s mood, which is never reset between Phases. Each denouncement or hostile choice moves it one step further, from unannoyed, to `cop_annoyed`, to `cop_very_annoyed`. The Corpse Phase’s endings read back a state the Player may have started building three Phases earlier. The second is the cauldron explosion. It must be reachable from any Phase, so it is declared once, in DURING PLOT, rather than being repeated inside Flower, Bone, and Corpse individually.

7.1.3 Results

7.1.3.1 Compilation

Table 7.3 reports the metrics of compiling *Everything For Science!*, all six source files, with the procedure and machine of the benchmarks (Section 7.2.1.2).

7.1.3.2 Observations

Authoring the game also produced several observations about using Regia in practice.

The game’s vocabulary leans heavily towards Events: 31 Events, against 14 Actions and 10 Facts. Part of this imbalance comes from a limitation: unlike Actions, Events take no pa-

Metric	Value
Files generated	17
Input LoC	526
Output LoC	1075
Compilation time	16.6 ms
Write time	0.3 ms
Total time	16.8 ms
Peak RAM	0.59 MB

Table 7.3: Metrics recorded compiling *Everything For Science!* in full.

rameters. Three groups of Events differ only in the item they concern: `picked_up_flower`, `picked_up_bone`, and `picked_up_corpse`; `give_flower`, `give_bone`, and `give_corpse`; and `denounce_flower`, `denounce_bone`, and `denounce_corpse`. With parametric Events, these nine declarations could have been three, `picked_up(item)`, `give(item)`, and `denounce(item)`, with no loss of expressiveness.

Handling the Subplot also required the main Plot to step out of its way. By design, a Subplot’s Playbooks can carry plans with the same triggers as the main Plot’s, and `PRIORITY` lets the Subplot’s version take precedence while both are assigned. A Plot would then describe only its own default behaviour, without knowing that a Subplot exists. Because `PRIORITY` is not enforced (Section 5.3.5), this layering is not safe: nothing guarantees which of two co-assigned plans with the same trigger fires. Listing 7.2 shows the workaround: before starting the Subplot, the main Plot unassigns `CopFlowerPlaybook` and `DogMainPlaybook`, clearing its own plans out of the way.

Redundancy also appeared within individual plans. `IF` and `ELSE` branches compile to independent AgentSpeak plans, one per branch (Section 5.3.3). Regia lets an author write statements shared by every branch before them, but not after them: logic meant to run whichever branch fires must be repeated at the end of every branch. In the game, the `denounce_bone` handler closes all three of its branches with the same two statements, clearing the dialogue and offering only the exit option.

Two aspects of the language worked as intended. Isolating the donut theft as a Subplot kept its cost contained: the addition touched the main Plot in two places, the five-line branch of Listing 7.2 and the handler of Listing 7.3, and left every Playbook outside the Subplot unmodified. The Subplot’s two new Roles needed no new agents to play them. The Cop and Dog agents are reassigned new Playbooks for the Subplot’s duration; the only agent added is the Subplot’s Director, which the main Director creates at runtime (Section 7.1.1.1).

7.2 Transpiler Benchmarks

The AgentSpeak a Regia program compiles to is not written by hand, and its size and shape depend on the transpiler’s translation choices. This section studies how compilation scales, and how each construct shapes the resulting AgentSpeak. The measurements support one claim: the cost of compiling a Regia program follows the size of the program an author writes, not the size or shape of the AgentSpeak it expands into. Section 7.2.1 describes the benchmark generator; Section 7.2.2 scales each construct alone; and Section 7.2.3 scales several together, as a project would grow over time.

7.2.1 The Benchmark Tool

To automate testing, we built a benchmark generator, separate from the transpiler, that produces synthetic Regia programs. It takes a configuration specifying how many instances of each construct to generate, among them Roles, Phases, Playbooks, Plans, branches, statements, and Subplots, and assembles a valid Regia program matching it. The program then passes through the ordinary transpiler pipeline, and the benchmark records a fixed set of metrics during the run. The generator, the measurement scripts, and the data of this section are available in `language/benchmarks/` of the repository [59], so that every sweep can be reproduced.

Table 7.4 lists every dimension the generator can scale, what it adds to the Regia source and to the emitted AgentSpeak, and the average number of AgentSpeak lines one instance adds. Phases have no such figure. A Phase produces output only through what is written inside it, and the generator fills each Phase by assigning a Playbook to every Role on entry. The cost the generator attributes to a Phase is therefore proportional to the number of Roles in the Plot: a property of the generator’s convention rather than of Regia, under which the measurements for Phases below should be read.

7.2.1.1 Monitored Metrics

Four metrics are recorded for every run: compilation time, with the time to write the output to disk recorded separately; peak memory during compilation; the *expansion ratio*, the number of generated AgentSpeak lines per line of Regia; and the number of AgentSpeak files emitted. Time and memory are measured in two separate compilations of the same program, since tracing memory allocations slows compilation down. Every configuration is run five times, and every figure in this section is the mean of the five runs.

To measure how cost scales, we fit a straight line by ordinary least squares to the logarithm of compilation time against the logarithm of source size, one point per configuration. Its

Dimension	Regia Effect	AgentSpeak Effect	AgentSpeak Lines per Instance
Actions, Events, Facts	Added to the shared vocabulary, then referenced throughout the generated Playbooks.	Declarations themselves produce no output; only usage populates plan triggers, guards, and bodies.	0
Roles	Declared in the Plot header; may be assigned a Playbook via ASSIGN in any Phase.	One <code>.asl</code> file per Role.	$29 + 1$ for each included Playbook
Phases	Declared sequentially in the Plot header; may contain ON ENTER/ON EXIT blocks and WHEN handlers.	on_enter/on_exit plans in the Director file, sized by what an author writes inside them.	depends on how many ASSIGN/UNASSIGN statements are written (see below)
Playbooks	Declared as standalone bundles, assignable to any Role.	One <code>.asl</code> file per Playbook: a fixed header, plus its Plans.	8 (header only; Plans below add their own cost)
Plans (per Playbook)	WHEN blocks added to a Playbook.	Base reactive plans inside the Playbook's file.	$2 + 1$ per statement
Branches (per Plan)	IF/ELSE blocks added to a WHEN body.	Each branch compiles to a separate AgentSpeak plan, as in Section 5.3.3.	$2 + 1$ per statement
Statements (per Branch)	Sequential DO statements added to a branch body.	Directly extends the corresponding plan body.	1, exact
Subplot Breadth	Controls the number of Subplots spawned in each Plot	One Director and Role file pair per sibling.	One full file pair per branch (linear in Breadth)
Subplot Depth	Additional nesting levels; each level's siblings themselves spawn Breadth-many children.	Each level multiplies the subplot count by the branching factor.	node count grows as $\text{breadth}^{\text{depth}}$ (exponential)

Table 7.4: Benchmark dimensions and their intrinsic cost, that is, the cost following from Regia's compilation rules alone, independent of any particular benchmark configuration. Phases carry no flat figure, since their cost depends on how many assignment statements an author writes inside them.

slope k is the exponent of a power law, $\text{time} \propto \text{size}^k$: $k = 1$ indicates linear growth, and $k > 1$ growth faster than linear. We report a 95% interval for k , derived from its standard error, and the coefficient of determination R^2 , computed on the logarithmic scale, which measures how closely the points follow such a law. A fixed startup cost weighs most on the smallest configurations, so k slightly understates how fast the cost of larger programs grows.

7.2.1.2 Test Environment

Table 7.5 lists the specifications of the machine every benchmark in this section was run on, reported for reproducibility.

Component	Specification
Processor	AMD Ryzen 9 7900X (12 cores, 24 threads)
Memory	30.44 GB RAM
Storage Type	NVMe SSD
OS	Arch Linux 7.2.3-arch1-2, x86_64
Runtime	Python 3.14.7

Table 7.5: Specifications of the machine used to run every benchmark reported in this section.

7.2.1.3 Default Configuration

Every sweep varies one or more dimensions against a fixed default configuration: 10 Actions, 10 Events, 5 Facts, 2 Roles, 2 Phases, and 2 Playbooks with 2 Plans each, 1 branch per Plan, and 2 statements per branch, with no Subplots. This baseline is close to the scale of `RoyalBanquet`, the running example of Chapter 5, with 2 Roles, 3 Phases, and 1 Playbook: an ordinary small program, not a stress case. Appendix B lists it in full, and Table 7.6 shows its metrics.

Metric	Value
Files generated	5 (1 Director, 2 Role files, 2 Playbook files)
Input LoC	117
Output LoC	269
Expansion ratio	2.30×
Compilation time	1.6 ms
Peak RAM	0.13 MB

Table 7.6: Metrics recorded for the default benchmark configuration.

7.2.2 Scaling Results

This section measures what the cost of each construct amounts to in practice, scaling one dimension at a time against the default configuration. Each dimension was swept across a range suited to its cost, with cheap constructs, such as statements per branch, carried further than expensive ones, such as Subplot Depth. Table 7.7 reports each range, with the input size, output size, and compilation time at its two ends. Actions, Events, and Facts are left out of this analysis, since an unreferenced declaration produces no output.

Dimension	Range	Input LoC	Output LoC	Compile time
Roles	1 → 1000	110 → 7,103	229 → 40,189	1.9 → 133.5 ms
Phases	1 → 200	102 → 3,087	257 → 2,645	1.7 → 45.1 ms
Playbooks	0 → 250	64 → 3,589	191 → 6,717	1.0 → 57.6 ms
Plans per Playbook	1 → 200	105 → 2,493	251 → 3,833	1.7 → 42.9 ms
Branches per Plan	0 → 50	105 → 705	253 → 1,249	1.6 → 13.2 ms
Statements per Branch	1 → 500	113 → 2,109	265 → 2,261	1.8 → 38.1 ms
Subplot Breadth	0 → 50	117 → 1,028	269 → 8,421	1.9 → 12.8 ms
Subplot Depth	0 → 6	117 → 2,768	269 → 20,933	1.9 → 34.1 ms

Table 7.7: Each benchmark dimension scaled alone against the default configuration: the range swept, and the input size, output size, and compilation time measured at each end of that range. Every figure is the mean of five runs.

Compilation time grew close to linearly with the source size produced by every dimension, with exponents between 0.93 and 1.17, and peak memory followed it. Four dimensions grow slightly faster than linearly: Roles, Plans per Playbook, Branches per Plan, and Statements per Branch. Subplot Depth is the one dimension whose cost grows exponentially in its own parameter, from 1.9 ms at depth 0 to 34.1 ms at depth 6. The cause is the source, not the output: the generator writes every Subplot as a Plot of its own, and each nesting level multiplies the number of Plots by the branching factor, so the source grows exponentially with depth. Relative to that source, the two Subplot dimensions are the cheapest to compile, at about 12 ms per thousand lines, although they expand the most, since each Subplot compiles to a Director file and a Role file of its own (Section 6.1.4.2). At depth 6, the transpiler turns 2,768 lines of Regia into 20,933 lines of AgentSpeak across 257 files, in 34 ms.

Comparing the dimensions shows what controls the cost. Pooling every point of the eight sweeps, the regression of compilation time against Regia source size gives an exponent of 1.03, with a 95% interval of 1.005 to 1.062, and a close fit ($R^2 = 0.99$): Figure 7.2 shows the eight dimensions falling together on one line. Against generated AgentSpeak size, the same regression fits far more loosely ($R^2 = 0.79$), since a line of Regia expands into different amounts of AgentSpeak depending on its construct, while costing about the same to compile. Per thousand lines of Regia, the eight dimensions span 12.3 to 18.8 ms at the top of their ranges.

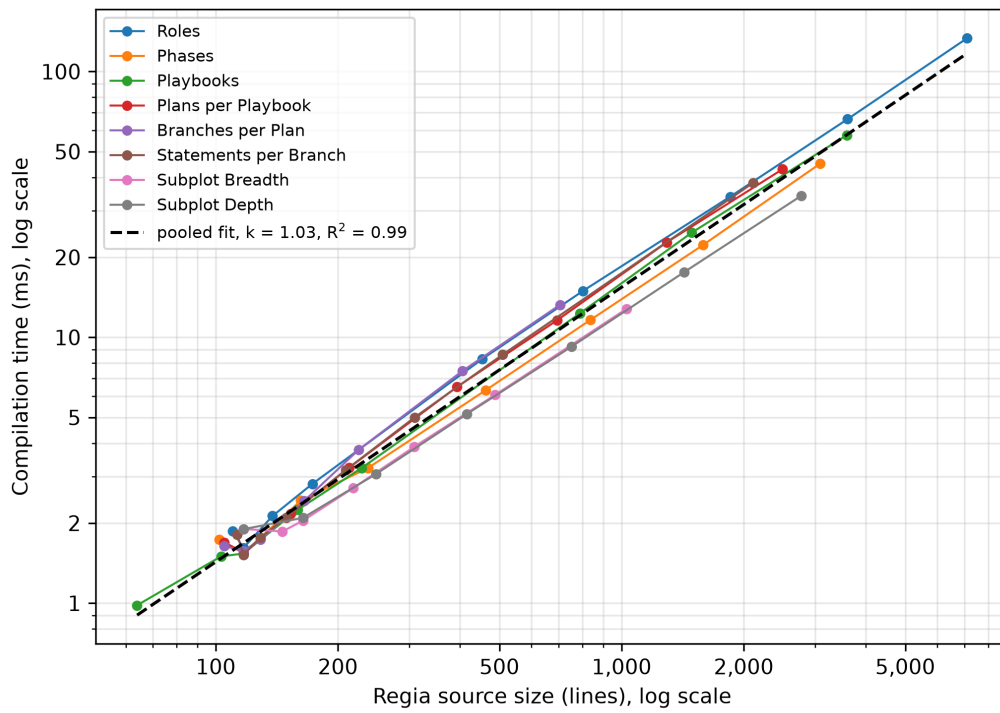


Figure 7.2: Compilation time against Regia source size, on logarithmic axes, for all eight benchmarked dimensions. The dimensions fall together on one line, with a pooled exponent of 1.03.

The compiler’s work is therefore controlled by the program an author writes, not by the program it emits. This follows from the pipeline of Chapter 6: five of its six phases read, build, and check the input, and only the last writes output. A construct that expands into much AgentSpeak costs no more to compile than one that expands into little.

The expansion ratio makes the same point from the other side. It depends on the constructs a program is made of, not on the language: scaling Roles reaches $5.66\times$ and Subplot Breadth $8.19\times$, since each produces whole files of largely fixed content, while scaling Statements falls to $1.07\times$ and Phases to $0.86\times$, fewer lines of AgentSpeak than of Regia. The $2.30\times$ of the default configuration is one point in this range, not a general figure for the language.

7.2.3 Scaling Interactions

The dimensions of Section 7.2.2 were scaled in isolation. Two further scenarios test whether costs compound when several dimensions grow together, reflecting two ways a Regia project could grow.

7.2.3.1 A Realistic Project, Scaled

The first scenario reproduces the profile of the case-study game of Section 7.1, with 3 Roles, 3 Phases, 10 Playbooks, and 4 Plans per Playbook, and scales these four components proportionally, since they would grow most over a game’s development. We measure projects two, five, ten, and twenty times the size of the case study; Table 7.8 reports the five points. The $1\times$ point compiles faster than the game itself, in 7.1 ms against 16.6 ms (Table 7.3): it reproduces only the game’s size profile, not its content, and is a single generated source with no imports to read and resolve.

Scale	Source LoC	Output LoC	Compile time
$1\times$ (profile reconstruction)	433	877	7.1 ms
$2\times$	1,284	2,238	20.9 ms
$5\times$	7,041	10,965	125.5 ms
$10\times$	27,316	40,990	568.0 ms
$20\times$	107,916	159,090	3.00 s

Table 7.8: Source size, output size, and compilation time at each tested scale of the realistic-project scenario, reconstructed from *Everything For Science!*’s profile.

Figure 7.3 plots compilation time against source size on logarithmic axes, with a dashed reference line extending the linear rate of the $1\times$ point. The regression across the five points gives an exponent of 1.09, with a 95% interval of 1.02 to 1.16. Scaling the four constructs

together therefore grows slightly faster than linearly, although the interval overlaps that of the pooled single-dimension sweeps. Cost per source line rises from 16.5 ms per thousand lines at the case-study profile to 27.8 ms at twenty times that size, a factor of 1.7 over a 250-fold increase in source size, while peak memory grows from 0.7 MB to 209 MB.

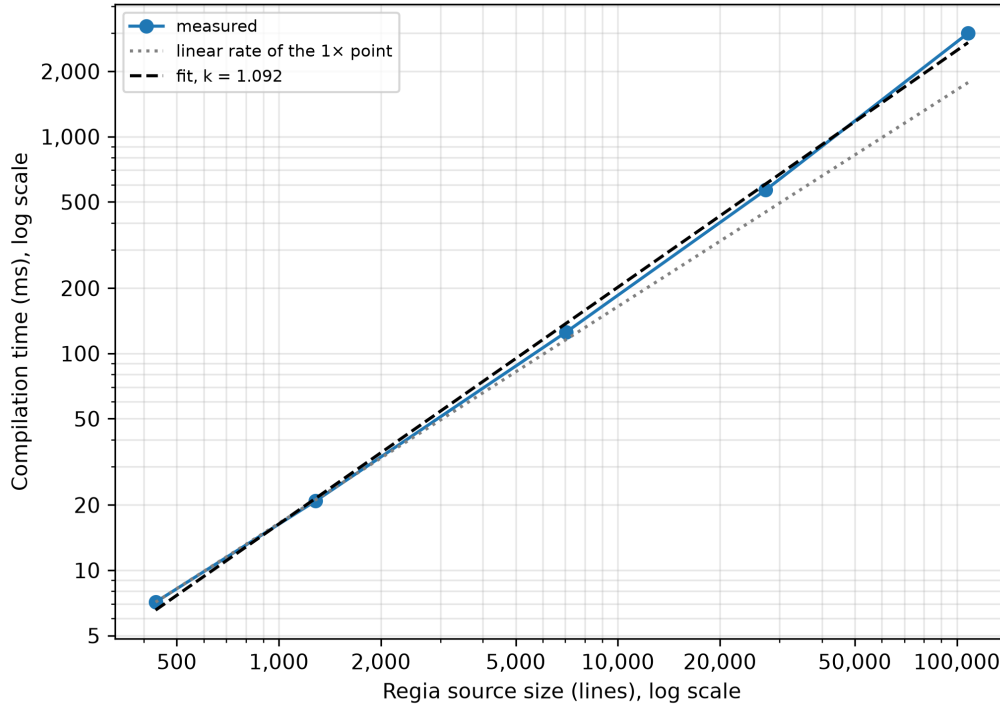


Figure 7.3: Compilation time against Regia source size, on logarithmic axes, for the realistic-project scenario, against a dashed reference line extending the 1× point’s own linear rate. The measured curve pulls slightly above the reference as the project grows, a growth exponent of 1.09.

The largest configuration is far beyond a realistic project: 107,916 lines of Regia, compiling to 159,090 lines of AgentSpeak across 263 files, in 3.0 s.

7.2.3.2 Construct Mix at Matched Size

The second scenario compares two compositions at equal size. The *narrative-heavy* configuration scales Roles and Phases together, with Playbook content fixed at the default, describing a plot-driven system. The *reactive-heavy* configuration scales Playbooks and Plans together, with Roles and Phases fixed, growing local behaviour. Since their input parameters cannot be matched, we compare the two curves at matched source size, interpolating between measured points where needed. Table 7.9 and Figure 7.4 report compilation cost per thousand lines of Regia.

Source size (LoC)	Reactive-heavy (ms/1k lines)	Narrative-heavy (ms/1k lines)
~300	16.3	15.9
~1,000	17.3	17.8
~4,000	18.5	18.9

Table 7.9: Normalised compilation cost for the reactive-heavy and narrative-heavy construct mixes, at three matched source sizes, interpolated from the measured sweep points.

Across the source sizes both sweeps cover, from 100 to about 8,000 lines, the two curves stay between 0.8% and 10.9% apart, and cross at about 350 lines, so neither composition is consistently the cheaper. Normalised cost rises by a factor of 1.24 for narrative-heavy and 1.19 for reactive-heavy. The two compositions behave almost identically: which constructs a program is built from does not determine what compiling it costs.

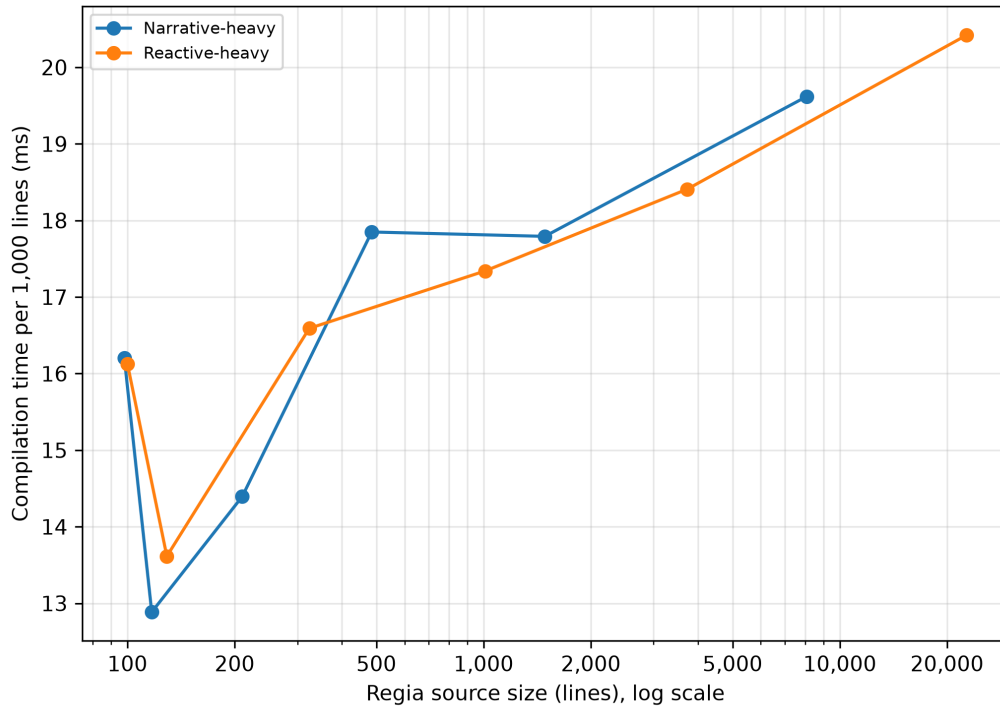


Figure 7.4: Normalised compilation cost against Regia source size, for the reactive-heavy and narrative-heavy construct mixes. The two curves cross at about 350 lines, and stay within 11% of each other across the range measured.

Both scenarios grew close to linearly in the size of the program an author writes. Scaling four constructs together gave an exponent of 1.09, slightly above linear, and the two construct mixes stayed within 11% of each other at matched source size. The largest configuration, twenty times the size of a complete game, compiled in 3.0s within 209 MB. Section 9.1 returns to these results.

Chapter 8

Language Evaluation

This chapter evaluates Regia against two external reference points. The first is a reference metamodel for multi-agent systems, which measures how much of the agent domain the language is able to express. The second is the people the language is intended for, whose understanding of Regia code we assess through a questionnaire. Section 8.1 describes the framework both draw on and how we depart from it, Section 8.2 reports the metamodel comparison, and Section 8.3 the questionnaire.

8.1 Evaluation Approach

Regia is, at its core, a domain-specific language for multi-agent systems. Alaca et al. [2] observe that most languages of this kind are presented with examples of their use but never assessed, and propose AgentDSM-Eval to address this: a framework for evaluating a multi-agent system DSML along two independent dimensions. The first is a semi-automatic comparison of the language’s abstract syntax against FAML [6], a reference metamodel for Agent-Oriented Software Engineering methodologies, which yields a coverage figure comparable across languages. The second is a multi-case empirical study, in which developers build several multi-agent systems with the language, report the time each stage took, and answer a questionnaire of twenty-four quality measures grouped under six characteristics, themselves adapted from the FQAD framework for assessing domain-specific languages in general [30].

We adopt AgentDSM-Eval as our reference but depart from the published procedure in four respects. First, we perform the metamodel comparison manually. The online tool that automated it is no longer in service, so each pairing was examined by hand.

Second, we score that comparison on three levels rather than two. AgentDSM-Eval records

a FAML concept as supported or unsupported; we distinguish a construct that realises a concept in full from one that realises part of it, and weight the latter accordingly. Section 8.2.1 gives the scoring. We discussed this choice with the authors of AgentDSM-Eval, who considered it an appropriate adaptation.¹ For fairness’s sake, we report coverage both with and without partial matches.

Third, our questionnaire adapts the procedure of the framework to our situation. AgentDSM-Eval considers trained users, people who have been taught the language and have already built systems with it. Neither we nor our collaborators at Untold Games had the time to carry out so thorough a procedure. Our respondents were instead given only a brief description of Regia in preparation for the task. Moreover, the questionnaire contains additional comprehension and code-writing tasks absent from the original. They also let us evaluate how quickly the language can be learnt, a crucial aspect for Regia, to which the original framework gives less prominence. Section 8.3 describes the instrument and what this difference means for the results.

Fourth, we do not carry out the framework’s multi-case development-time and throughput study for analogous reasons. That dimension requires several teams of agent developers to implement the same systems twice, once with the language and once without, which was beyond the resources available. Chapter 7 already addressed some of the related topics.

8.2 Comparison with the FAML Reference Metamodel

8.2.1 Method

FAML separates its concepts into two categories, design-time and runtime, mirroring the distinction between a class and an instance of that class. Twenty-six concepts belong to the first and twenty-one to the second. We examined every construct of Regia detailed in Chapter 5 against each of these forty-seven concepts, and scored each pairing.

We classified each FAML concept against the closest Regia construct assigning three possible scores. A concept received Full support, scoring 1.0, when a Regia construct existed with matching semantics. It received Partial support, scoring 0.5, when a Regia construct realised part of the concept’s semantics but omitted a defining property, for instance a typed declaration or an access-control rule. It received None, scoring 0.0, when Regia had no corresponding construct at all. We computed the coverage of each layer as the sum of scores divided by the number of concepts in that layer, and the overall coverage as the same ratio taken across all forty-seven concepts together. Tables A.1 and A.2 in the Appendix give the score and reasoning for every concept.

¹Private correspondence with the authors of AgentDSM-Eval, August 2026.

8.2.2 Results

Table 8.1 reports Regia’s coverage of the FAML metamodel, broken down by layer. Table 8.2 places these figures next to the coverage AgentDSM-Eval originally reported for the Prometheus Design Tool PDT [2].

Layer	Concepts	Full	Partial	None	Coverage (Full+Partial)	Coverage (Full Only)
Design-Time	26	4	6	16	26.9%	15.4%
Runtime	21	9	7	5	59.5%	42.9%
Overall	47	13	13	21	41.5%	27.7%

Table 8.1: Regia’s coverage of the FAML reference metamodel, by layer, with the concept counts behind each score and coverage computed with and without partial matches.

Language	Design-Time	Runtime	Overall
Regia (Full+Partial)	26.9%	59.5%	43.2%
Regia (Full Only)	15.4%	42.9%	29.1%
PDT (Prometheus)	52.0%	55.0%	53.5%

Table 8.2: Metamodel coverage of Regia and PDT compared, by layer. Overall is the mean of the two layers, the basis on which Alaca et al. [2] report PDT’s figures.

Regia matched Full or Partial support for 19.5 of the 47 FAML concepts, an overall coverage of 41.5%. Coverage is uneven across the two layers: 26.9% at design time against 59.5% at runtime, a gap of 32.6 percentage points. PDT shows the same direction of imbalance, but far less pronounced, 52.0% against 55.0%, a gap of 3 percentage points.

8.2.3 Interpretation

Averaged across the two layers, as Alaca et al. report PDT, Regia’s coverage is 43.2%, 10.3 percentage points lower than PDT’s 53.5%, and the gap lies at design time: counting partial matches, Regia’s runtime coverage of 59.5% exceeds PDT’s 55.0%. Counting full matches alone, Regia trails PDT in both layers, but by far more at design time, 15.4% against 52.0%, than at runtime, 42.9% against 55.0%.

Two clusters of FAML concepts account for most of the concepts scoring None, and both belong to the design-time group. First, Regia has no Requirements, Ontology, or Resource Layer. These higher-level concepts have no correspondence in the language. Second, FAML treats the environment as a first-class entity with typed, per-role sensed or changed properties, called Facets, a distinction Regia does not draw. Both gaps follow from the setting Regia targets, game development, where the environment and the objects in it are realised through a game engine, so an author has less need to model them in the language.

The design-time gap more generally follows from our deliberate design scope: Regia omits loops, arithmetic, variables, and user-defined functions, and restricts itself to Actions, Events, Facts, Playbooks, and Plots. Prometheus, the methodology PDT implements, spans the full development lifecycle, from system specification through architectural design to detailed design, and therefore models requirements, scenarios, organisation structures, and interaction protocols. FAML’s design-time layer is populated largely by concepts from that lifecycle stage, and Regia represents none of them, since it targets narrative behaviour authoring for an already partly-designed agent population, at a higher abstraction level.

Regia’s lower overall coverage therefore indicates a narrower scope, restricted to the narrative-orchestration and reactive-behaviour concerns Chapter 5 sets out, at the cost of the requirements and architecture-oriented concepts a general-purpose methodology such as Prometheus provides. The runtime figure is the one that speaks to the language’s own concerns, and there Regia reaches closer to PDT.

8.3 Questionnaire

The questionnaire tests the language with its users, rather than against a model. It adapts the qualitative instrument of AgentDSM-Eval, as Section 8.1 describes, and measures three things: whether respondents can read Regia, whether they can write it, and how they perceive its quality. The rest of this section follows that division throughout.

8.3.1 Instrument

Respondents first read a document introducing Regia through the running example of Chapter 5, and then were asked to answer twelve questions. They did not use the transpiler or the Visual Editor: both are run through Python, and we chose not to ask respondents without programming experience to install and operate them. The first question asked them to declare that they completed the form without help from other people or from generative AI tools. This ensures the significance of the recorded results. The second optionally recorded their technical background. Table 8.3 summarises the remaining ten.

The seven reading items each present a fragment of the running example, and asks a comprehension question. Three are multi-select: each lists five statements about the fragment, any number of which may be true. Four ask the respondent to explain in their own words what the fragment does, in as much detail as they can.

The generation task extends the running example, through the following specification. A Princess and a Pretender attend the banquet. The Princess greets the Guests at the reception, and at the feast refuses any food she does not like. The Pretender initially

Q	ID	Fragment	Format	Constructs tested	Measures
Q3	C1	insult_heard plan	Multi-select	PRIORITY, TEMPER, EFFECTS; branching; SIGNAL	Reading
Q4	O1	digestion_check plan	Explanation	a guarded body; FORGET; PRINT	Reading
Q5	O2	food_served plan	Explanation	a prefix statement; a com- pound condition	Reading
Q6	C2	RoyalBanquet header	Multi-select	PHASE ... INITIAL; ROLE	Reading
Q7	C3	DURING PLOT block	Multi-select	plot-wide handlers; WORLD DO; Role DO; END PLOT	Reading
Q8	O3	DURING reception block	Explanation	ON ENTER; ASSIGN; TRANSITION TO	Reading
Q9	O4	DURING feast block	Explanation	WHEN ROLE ... SIGNALS; START SUBPLOT; ON EXIT	Reading
Q10	–	A written specifi- cation	Code writ- ing, optional	fifteen specified behaviours	Writing
Q11	–	–	Likert scale	fifteen quality measures from AgentDSM-Eval	Perceiving
Q12	–	–	Free text	further comments	–

Table 8.3: The ten substantive questions, the fragment of the running example each presents, and what each measures.

behaves as any Guest, and may talk with the others. If no food pleases the Princess, she sends the Pretender to search for some, a quest the specification suggests realising as a Subplot. The Princess may also ask the Pretender to dance, starting a Ball during which the Pretender attends only to her. Guests greeted by the Princess become enamoured of her, and Guests spoken to by the Pretender become their friends; during the Ball, a friendly Guest cheers the couple and an enamoured Guest insults them, starting a duel. We decompose the specification into fifteen behaviours for scoring, listed in Figure 8.2.

The Likert grid retains fifteen of the twenty-four quality measures of AgentDSM-Eval, each rated on a five-point scale [34], from complete disagreement to full agreement, with an additional “I do not know” option. The nine omitted measures either presuppose a development environment or tool which the participants were not given, or assume familiarity with specific agent platforms and architectures, knowledge most of our participants did not have. As such, our ratings are not directly comparable with the scores Alaca et al. [2] report for PDT and the other languages they evaluate.

8.3.2 Participants and Procedure

The form was anonymous, distributed online to members of Untold Games and other volunteers. No time limit was imposed. Twenty-six people responded. Seventeen report a background in computer science alone, six combine it with game design or development, and three report a background in game design or development without computer science. The sample is therefore composed mostly of programmers.

Of the twenty-six participants, ten attempted the optional generation task. Some of these answers were cut short because of the 4000 character limit in the form’s textbox. We decided to still grade the available fragments. No response was excluded from the results.

8.3.3 Metrics

Throughout, we measure the relationship between two quantities with Spearman’s rank correlation coefficient ρ [56], which depends only on the order of the values. It suits ordinal Likert ratings, and makes no assumption about the distribution of the scores, which cluster near the maximum. Where we give an interval for ρ , it is a 95% percentile bootstrap interval [14]: the respondents are resampled with replacement 10,000 times, ρ is recomputed on each resample, and the interval spans the 2.5th to the 97.5th percentile of the results. Resamples in which either variable is constant, and ρ is therefore undefined, are skipped, and a fixed random seed makes the interval reproducible. These analyses are descriptive. The questionnaire was not designed around hypotheses fixed in advance, and its small, self-selected sample does not sufficiently represent the audience Regia is built for,

so we run no hypothesis tests and report no p -values.

8.3.3.1 Reading

Each multi-select item presents five statements, any number of which may be true, and the respondent selects those they judge to be true; a statement left unselected is thereby judged false. No single scoring rule for items of this kind is agreed in the literature [54]. We score them with normalised plus/minus scoring, which awards credit for each correct selection and deducts it for each incorrect one, and which Betts et al. [5] find among the most suitable rules for items asking respondents to select all statements that apply.

For one item, let n_T and n_F be its numbers of true and false statements, and, for one respondent, let TP be the number of true statements they selected and FP the number of false statements they selected. Each correct selection earns $1/n_T$, and each incorrect selection costs $1/n_F$:

$$S = \frac{TP}{n_T} - \frac{FP}{n_F}. \quad (8.1)$$

S ranges from -1 to 1 . A respondent who selects exactly the true statements scores 1 ; one who selects every statement, or none, scores 0 . The score is defined for every item, since each contains at least one true and one false statement.

Normalising each class by its own size matters because the three items have unbalanced keys, with three, two, and one true statements respectively. Under the unnormalised rule, one point per correct selection less one point per incorrect selection, a single wrong selection on Q7 cancels the points gained through its only true statement. Scoring each statement as right or wrong has the opposite flaw, rewarding inaction: selecting nothing on Q7 still judges four of its five statements correctly. Equation 8.1 gives no credit to either strategy, whatever the balance of the key.

The score coincides with Youden's J [64], a standard method for computing accuracy. The first term of Equation 8.1 is the sensitivity Se , the share of true statements selected, and the second is one minus the specificity Sp , the share of false statements left unselected, so that $S = Se + Sp - 1 = J$. S can therefore equally be read as the rate of correct selections less the rate of incorrect ones, the form in which a score of 0 marks performance no better than selecting at random.

Each explanation item has a rubric of six or seven units, each describing one point a complete answer covers. A unit is coded 2 if the answer covers it thoroughly, 1 if it covers it in less detail, and 0 if it is absent. With U the number of units in the item and c_u the code of unit u , the item's score is the sum of its codes over the maximum possible:

$$s = \frac{1}{2U} \sum_{u=1}^U c_u, \quad (8.2)$$

so that items with more units do not weigh more in an average. A single rater coded every answer, so no measure of agreement between raters is available; Section 8.4 discusses the consequence.

8.3.3.2 Writing

The generation task is scored with two rubrics. The first codes each of the fifteen behaviours on the same 0, 1, 2 scale, recording separately whether a behaviour scored 0 was attempted incorrectly or not attempted at all. Three measures follow from it: coverage, the score of Equation 8.2 taken over all fifteen behaviours; the number of behaviours attempted; and quality when attempted, the same score taken over the attempted behaviours alone.

We take quality when attempted as the primary measure. Coverage is dominated by length: it correlates with the number of lines a respondent wrote at $\rho = 0.72$. The form's length limit truncated only three submissions, and these were the ones most complete. Coverage therefore records how much a respondent wrote as much as how well they wrote it.

The second rubric counts errors, classified into the five categories of Table 8.5. We report them both as totals and per hundred non-blank lines, since a longer submission offers more opportunity for error.

8.3.3.3 Perceiving

For each quality measure, we exclude “I do not know” from the denominator and report it separately, since it records unfamiliarity with the measure rather than an opinion about the language. With n the number of remaining responses, and n_{SA} , n_A , n_D , and n_{SD} the numbers who strongly agree, agree, disagree, and strongly disagree, net agreement is

$$NA = \frac{n_{SA} + n_A - n_D - n_{SD}}{n}, \quad (8.3)$$

which ranges from -1 , unanimous disagreement, to 1 , unanimous agreement. We report medians rather than means, since the scale is ordinal.

8.3.4 Results

8.3.4.1 Reading

Respondents read Regia well. The median comprehension score is 0.84, with an interquartile range of 0.74 to 0.89. The median plus/minus score on the multi-select items is 0.81, and the median rubric score on the explanation items is 0.85. Figure 8.1 shows each respondent’s score on each item. Scores concentrate near the maximum on every item, and the Playbook-level and Plot-level items score alike, so neither level of the language is harder to read in general.

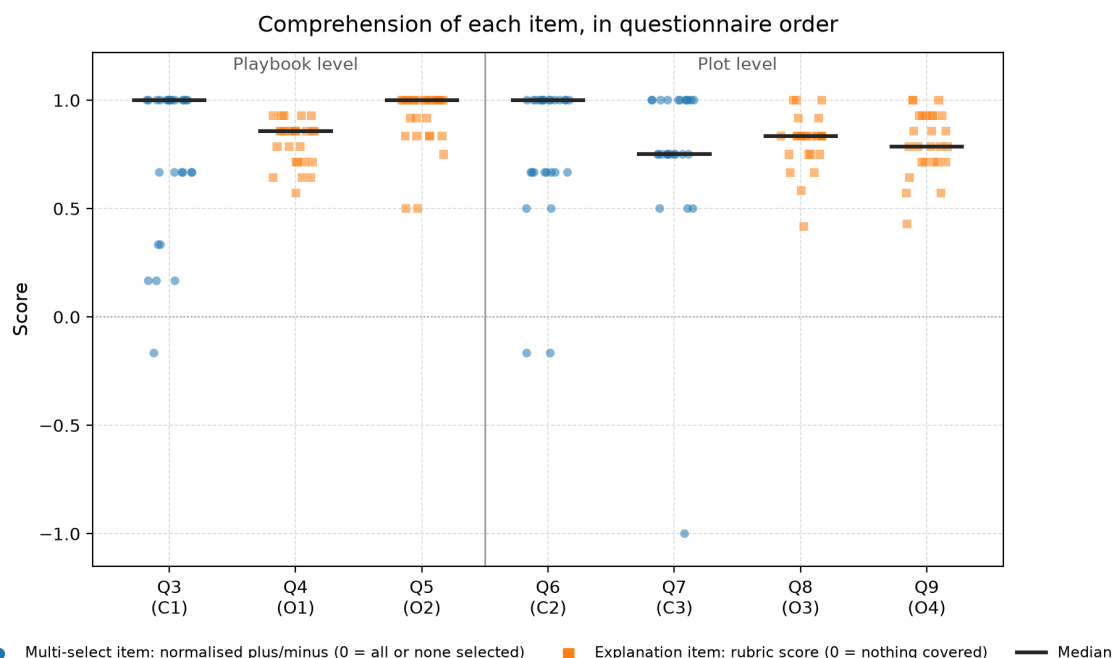


Figure 8.1: Each respondent’s score on the seven reading items, in questionnaire order, with the median of each item. Multi-select items report the normalised plus/minus score of Equation 8.1, for which 0 is the score of selecting every statement or none; explanation items report the rubric score of Equation 8.2, for which 0 is an answer covering nothing.

The errors that do occur are not scattered. Table 8.4 gathers the lowest-scoring statements and rubric units, which fall under two themes, each supported by both formats.

The first theme is what a Role denotes. More than a third of respondents read **ROLE Host**, as naming a single agent, and as many read two Role declarations as two agents. The explanation answers show the same reading independently. The lowest-scoring rubric unit in the questionnaire concerns whether a Playbook applies to every agent holding it, and the units on **ASSIGN** and **MAPPING**, both of which act on Roles, also score low. The second theme is when a block is active. Of the respondents, 31% read **DURING PLOT** as a phase

Theme	Evidence	Item	Result
What a Role denotes	False: “Host is one specific agent”	Q7	selected by 35%
	False: “Exactly two agents exist, Host and Guest”	Q6	selected by 35%
	A Playbook applies to every agent holding it	Q4	0.96 of 2
	ASSIGN acts on every agent playing the Role	Q8	1.62 of 2
	MAPPING maps one Role onto another	Q9	1.38 of 2
When a block is active	False: “PLOT is a phase”	Q7	selected by 31%
	A WHEN handler stays live throughout its phase	Q8	1.23 of 2
	A DURING block scopes behaviour to one phase	Q8	1.46 of 2
	ON EXIT fires on leaving the phase	Q9	1.27 of 2

Table 8.4: The statements most often misjudged and the rubric units scored lowest, grouped by the aspect of the language each concerns. Multi-select results give the share selecting a false statement; explanation results give the mean code of the unit.

named PLOT, and the units on whether a **WHEN** handler stays live throughout its phase, and on when **ON EXIT** fires, are among the lowest scored.

Individual statements, by contrast, read well. Every rubric unit concerning an action, a condition, **FORGET**, **PRINT**, or **START SUBPLOT** has a mean of 1.7 or above. This includes the compound condition of Q5, read with the correct precedence by nearly every respondent. The difficulty lies in Regia’s structural abstractions rather than in its statements.

8.3.4.2 Writing

Ten respondents wrote an extension of the running example. Figure 8.2 shows how many implemented each behaviour.

What respondents wrote was mostly correct. Quality when attempted has a median of 0.69, and every respondent falls between 0.60 and 0.91. How much they wrote varied far more: from four behaviours to all fifteen, and from 7 to 138 lines. Coverage falls steadily down the specification, which lists its simpler behaviours first. Nearly every respondent declared the two new Roles and implemented their behaviour at the reception and the feast, while the Ball and the Guests’ reactions to it were mostly not attempted. The form’s length limit cut short three submissions, all three of which had reached the final behaviours.

Table 8.5 classifies the 166 errors found across the ten submissions.

Four-fifths of the errors concern syntax or undeclared vocabulary, and both are errors the

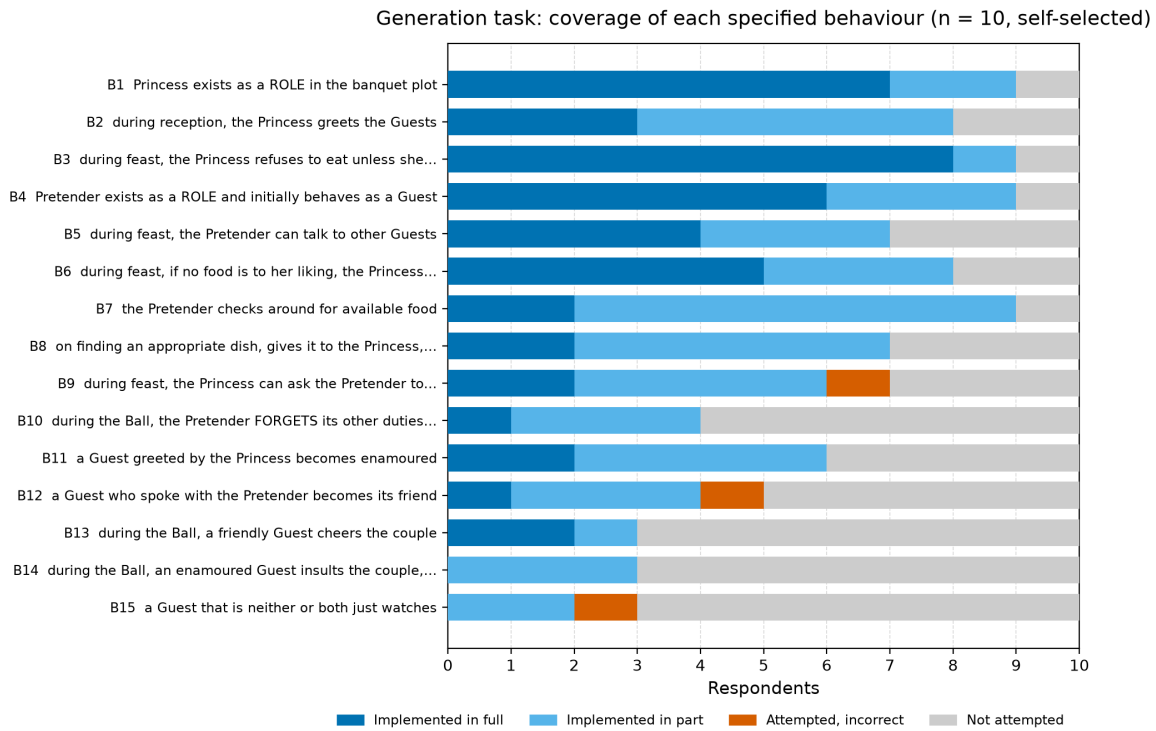


Figure 8.2: Coverage of each of the fifteen specified behaviours across the ten respondents who attempted the generation task, in the order the specification lists them.

Error category	Share	Submissions
Pure syntax: missing full stops, omitted DO, identifier case, block structure	53.6%	10 of 10
Vocabulary used without being declared	27.1%	8 of 10
Invented or malformed construct	14.5%	6 of 10
Vocabulary category confused, such as an Event used as an Action	3.6%	4 of 10
Plot-level construct inside a Playbook, or the reverse	1.2%	1 of 10

Table 8.5: Errors in the generation task by category, as a share of all 166 errors, and the number of submissions containing at least one.

transpiler reports: undeclared vocabulary is the case Section 6.1.5 uses as its example diagnostic. Errors placing a construct at the wrong level of the language occurred twice, in a single submission, and errors confusing Events, Actions, and Facts six times. Respondents writing Regia kept local behaviour and global narration apart, and kept the three vocabulary categories distinct. The invented or malformed constructs record something different: constructs respondents reached for that Regia does not provide in the form they used, such as a **WHEN** handler with a constant condition. Errors are also concentrated: a single submission contributed 24 of the 89 syntax errors.

Self-selection into the task appears small. Respondents who attempted it and those who did not have the same median comprehension score, 0.84.

8.3.4.3 Perceived Quality

Figure 8.3 shows the distribution of responses on the fifteen quality measures.

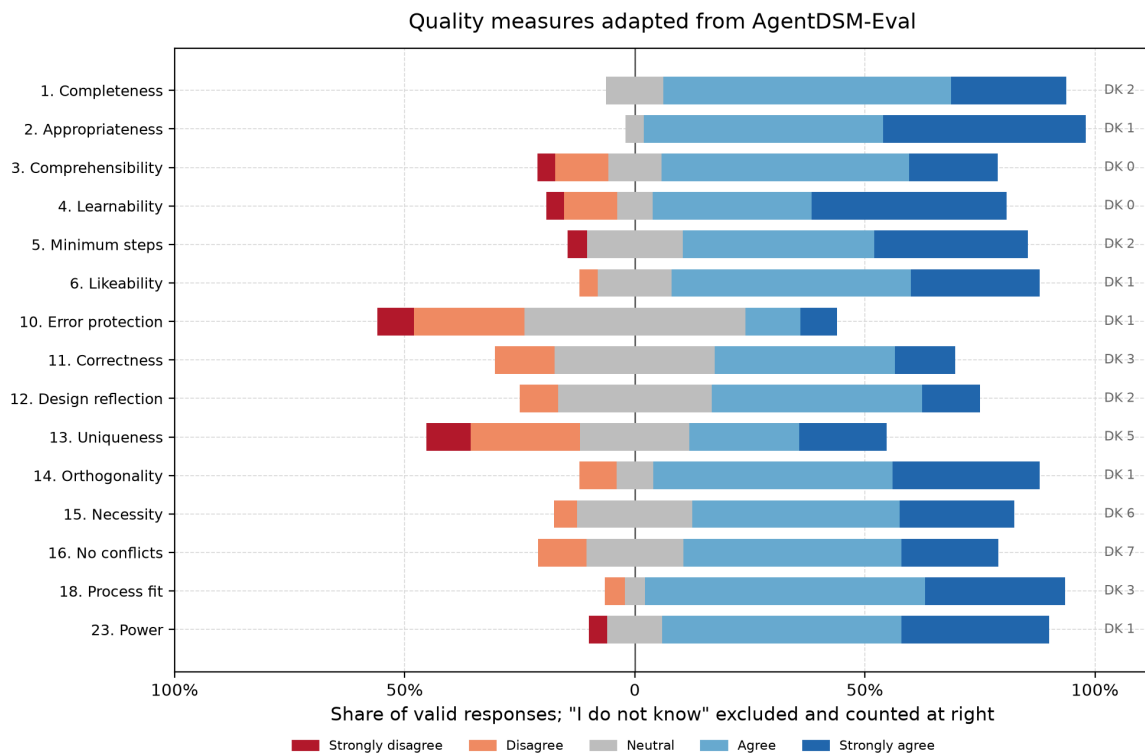


Figure 8.3: Responses to the fifteen quality measures adapted from AgentDSM-Eval, centred on the neutral response. “I do not know” is excluded from each bar and counted at the right.

Respondents rated Regia positively on fourteen of the fifteen measures. Grouped by the characteristics of AgentDSM-Eval, net agreement is +0.92 for Functional Suitability, +0.87

for Compatibility, +0.80 for MAS Development, +0.66 for Usability, +0.52 for Expressiveness, and +0.13 for Reliability. Comprehensibility and Learnability score +0.58 and +0.62, below the two functional measures: respondents were more confident that Regia suits its domain than that it is easy to learn.

Comprehensibility is also the one perception measure that can be checked against performance. The nineteen respondents who agreed that understanding Regia requires little effort demonstrate a median comprehension of 0.86, against 0.76 for the seven who did not. Across all twenty-six, however, self-rated comprehensibility and measured comprehension correlate only weakly, at $\rho = 0.31$, with a 95% bootstrap interval of -0.05 to 0.60 . The interval includes zero, so this sample does not establish whether respondents' confidence tracks their understanding.

One measure falls below zero, and one only just above it. Error protection scores -0.12 , and accounts for the low Reliability figure. Respondents read a description of Regia and never compiled a program, so the validation and diagnostics of Chapter 6 were not visible to them; the measure records an aspect which would be mitigated in an actual development scenario. Uniqueness, whether Regia provides one and only one way to express each concept, scores +0.10. "I do not know" was most common on uniqueness, necessity, and the absence of conflicting elements, the three measures phrased in the most technical terms.

8.4 Threats to Validity

We state the threats to both evaluations here. For the questionnaire we follow the four categories of validity common in empirical software engineering [60].

8.4.1 Metamodel Comparison

A single analyst mapped Regia's constructs to FAML's concepts by hand, a judgement AgentDSM-Eval's own procedure also requires for every concept its tool cannot match [2], and distinguishing Full from Partial support adds a second judgement to each pairing. The comparison with PDT cannot be applied directly either, since PDT was scored by the framework's tool, in binary, by other evaluators. Reporting coverage with and without partial matches addresses both, and shows which conclusions depend on the partial judgements: the design-time gap holds under either scoring, while Regia's runtime coverage matches PDT's only when partial matches are counted.

8.4.2 Questionnaire

8.4.2.0.1 Construct validity. Respondents read Regia from a document its authors wrote, so comprehension scores measure the language and the document together. They never used the compiler, so ratings of error protection had little basis. A single rater coded every answer, so the codes' reproducibility is not established; all codes are in the repository for inspection (`q_scores/`). The form's length limit truncated the three most complete generation submissions, making their coverage a lower bound.

8.4.2.0.2 Internal validity. The form was unsupervised, so independence rests on each respondent's declaration, and its fixed order may have let fatigue lower scores on later items and on the generation task. One response, completed in seven minutes with the lowest comprehension score, is consistent with reading a multi-select instruction the wrong way round. We retain it, having set no exclusion rule in advance; removing it changes no conclusion. The five responses collected last were completed faster than the others, with a median of 62 minutes against 120, scored lower on the explanation items, with a median of 0.79 against 0.87, and include no attempt at the generation task. We retain them under the same rule. They account for the weak correlation between self-rated and measured comprehension: over the first twenty-one responses, $\rho = 0.53$, with a 95% bootstrap interval of 0.18 to 0.77.

8.4.2.0.3 External validity. Twenty-three of the twenty-six respondents have a background in computer science. The results therefore mainly describe programmers new to Regia, not the designers without programming experience for whom it is intended. Respondents were volunteers, and the reading items present short fragments rather than a narrative spread across many files.

8.4.2.0.4 Conclusion validity. With twenty-six respondents and scores near the maximum, correlations rest on few data points, so we report every result descriptively and compute no significance tests. The single correlation reported is given with its bootstrap interval, which includes zero: its direction is not established.

Two threats bear most on the conclusions: the runtime comparison with PDT depends on the partial judgements, and the questionnaire describes programmers rather than designers. Section 9.1 reads the results with both in view.

Chapter 9

Discussion and Conclusions

Chapter 7 put Regia to use, and Chapter 8 measured it through an evaluation framework. Neither answers on its own whether the language does what it was built to do. This chapter does. Section 9.1 first returns to the requirements of Chapter 4, then weighs the evidence of both chapters against the three concerns Chapter 3 found no existing system to address at once: accessibility to an author who does not program, deliberative reasoning, and embodiment in a game engine. The limits set out in Section 8.4 apply throughout. Section 9.2 states what the language and its tools do not yet do, and the work that follows, and Section 9.3 concludes.

9.1 Discussion

9.1.1 Requirements Revisited

Chapter 4 derived five requirements from our collaboration with Untold Games (Table 4.1). Table 9.1 gives the status of each and where the evidence for that judgement lies. Two are met in full. The three met in part fall short for different reasons, which the rest of this section takes up.

9.1.2 Accessibility

Programmers who had not seen Regia before read it well and wrote it mostly correctly (Section 8.3.4). Whether designers without programming experience can do the same is not established, since the sample consists mostly of programmers (Section 8.4). Within this audience, individual statements and the overall structure of the language were well

Requirement	Stated in	Status	Evidence
R1: Accessible syntax and vocabulary	Section 4.2	Partial	Syntax designed to it (Chapter 5); read and written well by programmers new to Regia (Section 8.3.4); not tested with designers (Section 8.4)
R2: A narrative-centred model	Section 4.3	Met	Plots, Roles, and Playbooks (Chapter 5); a complete game (Section 7.1); Role and Organization matched in full (Section 8.2)
R3: Dynamically assigned behaviour	Section 4.4	Partial	ASSIGN and UNASSIGN (Chapter 5); PRIORITY not enforced by the platform, forcing manual withdrawal (Section 7.1.3.2)
R4: A visual representation of the narrative	Section 4.5	Partial	The Visual Editor (Section 6.2); one Plot rendered, two edit operations (Section 6.2.4); not part of the questionnaire (Section 8.3.1)
R5: Modularity and reuse	Section 4.6	Met	Imports, Subplots, and Role mappings (Chapter 5); a Subplot added while touching the main Plot in two places (Section 7.1.3.2)

Table 9.1: The requirements of Chapter 4, whether each was met, and where the evidence for that judgement lies.

understood, but two concepts were misread more often than the rest: the Role abstraction, and how long a **WHEN** block stays active. The second is exactly the structure the Visual Editor draws, which the questionnaire did not test (Sections 6.2 and 8.3.1).

The Role is at once one of the language’s most effective constructs and the one readers misread most often. It is the one element that connects the global and local levels of the narrative. In *Everything For Science!*, it let the Cop and Dog agents move from the main Plot into a Subplot without new agents or changes to their Playbooks (Section 7.1.3.2). In the evaluation, however, a sizeable minority of respondents read a Role as naming a single agent (Section 8.3.4). The notation offers one possible explanation, since `ROLE Host` has the form of a declaration of a single entity. The misreading may equally arise from how the explanatory document introduced Roles, or from programming habits in which a declaration introduces one object, and the questionnaire cannot distinguish between these. In each case, the remedy lies in how a Role is introduced, in the language or in its documentation, and Section 9.2 returns to it.

Error protection was the quality measure respondents rated lowest (Section 8.3.4). Yet most of the errors in the generation task concern syntax, or vocabulary used without being declared (Table 8.5), and the transpiler reports both, with a location and a correction hint (Section 6.1.5). The weakness respondents perceived may therefore be partly one the tooling already addresses. The data cannot confirm this: no respondent used the transpiler, and whether its diagnostics help authors correct their programs has not been tested.

Section 4.1 asked that a designer be able to check that their intent survived the translation into code, which presupposes that a reader can tell whether they have understood. The evidence on this point is inconclusive. Respondents who judged Regia easy to understand scored higher than those who did not, but the correlation between self-rated and measured comprehension is weak, and this sample cannot establish its direction (Section 8.3.4).

9.1.3 Deliberation and Expressiveness

At run time, Regia exposes the primitives of the BDI model in full. FAML’s Belief, Event, Plan, Action, and Agent Goal concepts are all matched completely, the last through the **ACHIEVE** special action, which posts a goal for an agent to pursue (Tables A.1 and A.2). What Regia lacks is the declarative side of the model. FAML’s System Goal, Mental State Specification, and Mental State concepts have no counterpart: a plan can post a goal, but an author cannot declare a state that an agent, or the system as a whole, is meant to reach. Concepts of different nature take their place. A Plot’s Phases and the transitions between them determine where a narrative is heading, so the direction of the system is written as a sequence of story stages rather than as goals to achieve. This follows the decision of Section 4.3 to place the narrative at the centre of a program, and it has a cost: what an

agent pursues is left to Jason’s reasoning cycle, through what plans their Playbooks make available to them, while the author decides only when an agent’s behaviour changes.

One of the means an author has for making that decision does not fully reach the running program. **PRIORITY** ranks plans that respond to the same Event, but Jason’s default plan selection does not consume the annotation, as Section 5.3.5 notes. Respondents read its intended meaning correctly (Section 8.3.4), and building the case study showed the consequence of its absence. A Subplot was meant to override the main Plot’s plans through **PRIORITY** while both remained assigned; instead, the main Plot had to withdraw its own Playbooks explicitly, and so had to know that the Subplot existed (Section 7.1.3.2). Where this happens, what an author writes and what the program does diverge. The limitation lies in the platform rather than in the language, and Section 9.2 considers how it could be addressed.

Regia’s expressive gaps are of two kinds. Some follow from its scope, as Section 8.2.3 shows for FAML’s design-time concepts. Others were not intended, and surfaced independently. Building the game showed that, since Events take no parameters, nine of its 31 Events could have been three; and that, since **IF** and **ELSE** branches share no continuation, logic common to every branch must be repeated in each (Section 7.1.3.2). In the generation task, a share of the errors used constructs Regia does not provide in the form written (Table 8.5), recording what respondents expected to be able to write: forms Regia provides only in a narrower shape, or not at all. Widening them would remove some of the repetition the case study found, at the cost of the simplicity the language was designed for; Section 9.2 weighs that trade.

9.1.4 Embodiment and Practicality

A complete game written in Regia runs inside the Godot engine (Section 7.1). Regia reaches it through VEsNA rather than addressing embodiment itself, so what it contributes on this axis is a higher-level way to author what VEsNA already embodies. The toolchain needed to build a complete game does not appear to limit how large a Regia project can grow. Compilation cost follows the size of the source an author writes, close to linearly for every construct measured, and a project twenty times the size of the case study, 107,916 lines of Regia, compiles in 7.45 s within 209 MB (Sections 7.2.2 and 7.2.3).

What grows with a project is the work outside Regia. The transpiler stops at agent source (Section 6.1.4): each declared Action must be backed by a hand-written internal action, each agent must be declared in the deployment configuration, and each message must be handled in the engine. The 14 Actions of the case study each required a file of their own (Section 7.1.1). This cost is paid once per Action, so a project that grows by adding Plots, Playbooks, or Roles over an existing vocabulary adds none of it.

9.1.5 Regia Against Related Work

Chapter 3 found that no system it surveyed treats a scenario involving several independently reasoning agents as its own authorial unit, accessible to someone without a programming background, down to how each agent deliberates (Section 3.4). Regia addresses all three parts of that combination, with the qualifications established in the preceding subsections.

Scribe, the system Chapter 3 found closest to this combination, joins accessibility with narrative orchestration but draws its boundary explicitly at the behaviour of each character, which it leaves to separate systems. Regia places both on the same side of that boundary: a Plot orchestrates the narrative and a Playbook defines how each agent within it behaves, in one language that compiles to BDI agents.

9.1.6 Applicability Beyond Games

Regia was designed for games, but its model is not specific to them, and three common multi-agent scenarios show where it fits. For a single autonomous agent, such as a cleaning robot, Regia adds structure without benefit: with one participant and no scenario to coordinate, Plots and Roles have nothing to organise, and plain AgentSpeak describes the agent more directly. A market scenario, such as an auction, falls outside what Regia can express. Bidding requires an agent to compute and compare values, and Regia has no variables or arithmetic (Section 4.2); that reasoning would have to be delegated to Actions implemented outside the language. A coordinated search, such as a team of rescue drones looking for a missing person, matches Regia’s model closely. The mission divides into phases, from sweeping an area to converging on a sighting. Drones take Roles, such as scout and relay, and each Role receives the Playbook its phase requires, while handling a sighting can become a Subplot. The Director that sequences a narrative would sequence the mission in the same way. Since Regia compiles to standard Jason agents, such a system would run in any Jason environment, not only in Godot. We have not built these systems, so this assessment rests on the language’s design rather than on experience.

9.2 Limitations and Future Work

Several of the language’s gaps have direct solutions. First, we plan to remove the full stops that terminate statements, and to delimit blocks by indentation instead, closer to Python; this removes the most common class of syntax error in the generation task (Table 8.5). Events will be given parameters, in line with Facts and Actions. IF and ELSE blocks will be brought closer to their counterparts in general-purpose languages, which is

what respondents appeared to expect. The **MAPPING** clause will state how many members of a Role are mapped into a Subplot, making explicit what readers most often misread (Section 9.1.2). **PRIORITY** will be enforced, either through a custom plan selection function in Jason, analogous to the one VEsNA uses for **TEMPER**, or in the interpreter described below. New features are planned as well: a structure for dialogue, which every narrative language of Section 3.1.2 provides, the scheduling of actions, and a more direct expression of deliberation (Section 9.1.3).

Among the tools, the visual editor has the most room for improvement, and we plan to change the transpiler to integrate with it more closely. In particular, we plan to rewrite the transpiler in Rust. Python served it well while the language was changing quickly, which is no longer the case. A Rust transpiler could run alongside the editor in the browser, removing the dependency on an external interpreter. The rewrite would also build into the pipeline decisions reached during development, such as the optional emission of concrete syntax trees. This would let edits made on the canvas be written back through the compiler rather than by manipulating text (Section 6.2.3). The transpiler will also generate default agent configuration files, reducing the work left outside Regia (Section 9.1.4).

Other extensions are being developed within our department. A C++ AgentSpeak interpreter is the subject of Francesco Matano’s master’s thesis [36]. It would remove the dependency on the Java runtime, adapt more easily to Regia’s needs, and integrate more closely with game engines; a Godot plugin, letting the system communicate with the game through native signals, is being developed as part of the same work. Another project aims to translate natural-language descriptions of narratives into Regia code, further reducing the authoring burden.

A larger study is also needed, of the shape Section 8.4 implies: more participants, designers rather than programmers, working in the editor rather than only through text. Our collaboration with Untold Games will continue as the language develops, which makes such a study possible, and we plan to involve other studios, to increase the generality of our results.

9.3 Conclusions

Game designers describe characters by what they know, want, and have decided to do. The systems that bring those characters to life, however, are written by programmers, in terms designers cannot read. This thesis presented Regia, a domain-specific language in which designers author Belief-Desire-Intention agents at the level of the narrative. Plots describe scenarios, Roles the parts within them, and Playbooks the behaviour an agent receives while playing a part. Regia compiles to AgentSpeak for the Jason interpreter, comes with a visual editor for the structure of a narrative, and runs in the Godot engine

through VEsNA.

The evidence supports the approach within clear limits. A complete, branching game was written in Regia, and compilation cost grows almost linearly with source size, regardless of contents: a project twenty times the size of that game compiles in about three seconds. Regia covers the runtime concepts of a metamodel of agent-oriented methodologies well and its design-time concepts less, a consequence of centring the narrative rather than the agent. Respondents new to the language, mostly programmers, read and wrote it largely correctly, although many misread what a Role denotes. Of the five requirements, two are met in full and three in part. Above all, the language now needs a study with the designers it was built for. Regia shows that the narrative, rather than the individual agent, can be the unit a designer writes, while each character still deliberates on its own.

Appendix A

FAML Metamodel Mapping

Table A.1: Full mapping of the 26 FAML design-time concepts to Regia constructs, with FAML’s canonical definitions (Table 9 of Beydoun et al.).

FAML Concept	FAML Definition	Regia Mapping	Support	Reasoning
Action Specification	Specification of an action, including any pre-conditions and post-conditions.	ACTION declaration	Partial	Names and typed parameter slots exist; no pre-condition or postcondition can be declared.
Agent Definition	Specification of the initial state of an agent just after it is created.	(none)	None	No per-agent initial-belief or initial-state declaration construct exists.
Environment Statement	A Boolean statement about the environment.	FACT declaration / IF condition	Partial	Regia Facts are agent-scoped beliefs; the concept is conflated with Belief rather than kept as a distinct environment statement.

Continued on next page

Table A.1 – continued from previous page

FAML Concept	FAML Definition	Regia Mapping	Support	Reasoning
Facet Action Specification	Specification of a facet action in terms of the facet definition it will change and the new value it will write to the facet.	WORLD DO BELIEVE(fact) . (Director self-belief write)	Partial	A Director-held belief write is structurally facet-like, since it produces state shared above any single Role-playing agent, but it is not a typed, declared facet write.
Facet Definition	Specification of the structure of a given facet, including its name, data type and access mode.	(none)	None	No declaration distinguishes a shared, environment-level fact from an ordinary agent FACT , and no per-role sense or change permission exists.
Functional Requirement	Requirement that provides added value to the users of the system.	(none)	None	Regia has no requirements layer.
Interaction Protocol	Specification of patterns of communications that occur in the system.	(none)	None	TELL and BROADCAST exist as primitives, but no protocol or conversation structure can be declared.
Mental State Specification	Specification of the initial mental state in terms of specified beliefs and agent goals.	(none)	None	TEMPER declares personal-ity thresholds, not an initial belief or goal state.
Message Action Specification	Specification of a message action in terms of the message schema and parameters to use.	DO TELL (...). / DO BROADCAST (...).	Full	Direct match to the reserved TELL and BROADCAST primitives.

Continued on next page

Table A.1 – continued from previous page

FAML Concept	FAML Definition	Regia Mapping	Support	Reasoning
Message Schema	Specification of the structure and semantics of a given kind of messages that can occur within the system.	(none)	None	TELL and BROADCAST take untyped argument lists; no schema can be declared.
Non-Functional Requirement	Requirement about any limits, constraints or impositions on the system to be built.	(none)	None	Regia has no requirements layer.
Ontology	Structural model of a given domain.	(none)	None	The ACTION , EVENT , and FACT vocabulary is flat and untyped; no domain ontology construct exists.
Organization Definition	Specification of a collection of roles and agents co-operating towards a system goal.	PLOT header (PHASE and ROLE declarations)	Full	A Plot's declared ROLE set is exactly a collection of agents co-operating toward the Plot.
Plan Resource Specification	Specification of resources that are used in the Plan Specification.	(none)	None	Regia has no resource concept.
Plan Specification	An organized collection of action specifications.	WHEN block body (DO statement sequence)	Full	A WHEN body compiles to an ordered AgentSpeak plan body.
Policy	A rule that specifies an arrangement of events expected to occur in a given environment.	(none)	None	DURING and TRANSITION give structure to the Plot, but no first-class, declarative Policy object exists.
Requirement	Feature that a system must implement.	(none)	None	Regia has no requirements layer.

Continued on next page

Table A.1 – continued from previous page

FAML Concept	FAML Definition	Regia Mapping	Support	Reasoning
Resource Specification	A resource specification specifies something that has a name, may have reasonable representations and that can be acquired, shared or produced.	(none)	None	Regia has no resource concept.
Role	Specification of a behavioural pattern expected from some agents in a given system.	ROLE declaration	Full	Direct match.
Role Compatibility	Role relationship in which the source role is incompatible with the destination role for a given purpose.	(none)	None	No inter-role relationship can be declared.
Role Dependency	Role relationship in which the source role depends on the destination role for a given purpose.	(none)	None	No inter-role relationship can be declared.
Role Relationship	Social relationship between two roles for a given purpose.	(none)	None	MAPPING binds roles across a subplot call, but it does not declare a persistent relationship between them.
Service	A single, coherent block of activity in which an agent will engage.	PLAYBOOK	Partial	Playbooks bundle coherent behaviour, but they are not advertised or discoverable as services.

Continued on next page

Table A.1 – continued from previous page

FAML Concept	FAML Definition	Regia Mapping	Support	Reasoning
System	Final product of an agent-oriented software development project.	PLOT / whole .regia program	Partial	Regia has no single System wrapper distinct from an individual Plot.
System Goal	A specification of the state of the environment that the system tries to achieve.	(none)	None	The IF guard preceding a TRANSITION weakly resembles a goal condition, but no first-class goal-of-the-system object exists.
Task	Specification of a piece of behaviour that the system can perform.	ACTION declaration (conflated)	Partial	Regia draws no Task/Action distinction; both collapse into ACTION.

Table A.2: Full mapping of the 21 FAML runtime concepts to Regia constructs, with FAML’s canonical definitions (Table 10 of Beydoun et al.).

FAML Concept	FAML Definition	Regia Mapping	Support	Reasoning
Action	Fundamental unit of agent behaviour.	DO action (compiled AgentSpeak action call)	Full	Direct match.
Agent	A highly autonomous, situated, directed and rational entity.	Role-bound agent instance (compiled Role .asl running agent)	Full	Direct match.
Agent Goal	An environment statement which represents a state pursued by an agent.	DO ACHIEVE (goal). → !goal	Full	Direct match.

Continued on next page

Table A.2 – continued from previous page

FAML Concept	FAML Definition	Regia Mapping	Support	Reasoning
Belief	An environment statement held by an agent and deemed as true in a certain timeframe.	FACT instance / DO BELIEVE (fact).	Full	Direct match.
Communication	Composition of more than one message.	(none)	None	No conversation or communication object exists; only individual TELL and BROADCAST calls exist.
Environment	The world in which an agent is situated.	WORLD pseudo-actor	Partial	WORLD is only an action-call target, not a modelled Environment entity.
Environment History	The sequence of events that have occurred between the environment start-up and any given instant.	(none)	None	No event log construct exists.
Environment Statement	A statement about the environment.	FACT / condition	Partial	The same caveat as the design-time concept applies: Regia conflates this with agent Belief.
Event	Occurrence of something that changes the environment history.	EVENT declaration	Full	Direct match.
Facet	Property of the environment with which agents can interact.	Director-held belief (via WORLD DO BELIEVE)	Partial	The belief is structurally shared, above any single Role-playing agent, but no typed declaration and no per-role access control exist.

Continued on next page

Table A.2 – continued from previous page

FAML Concept	FAML Definition	Regia Mapping	Support	Reasoning
Facet Action	Action that results in the change of a given facet.	WORLD DO BELIEVE (fact). / WORLD DO FORGET(fact).	Partial	The statement functions as a shared-state write, but it is an untyped belief operation rather than a declared facet action.
Facet Event	Event that happens when the value of a facet changes.	(none)	None	The internal event AgentSpeak triggers on a belief addition stays inside the Director; it does not propagate to sensing Role agents.
Mental State	Agent goals and beliefs held by an agent at a certain timeframe.	(none)	None	TEMPER and EFFECTS model emotion and personality, not a queryable belief-and-goal state.
Message	Unit of communication between agents, which conforms to a specific message schema.	Arguments of TELL / BROADCAST	Partial	The payload exists ad hoc, with no backing schema.
Message Action	Action that results in a message being sent.	DO TELL (...). / DO BROADCAST (...).	Full	Direct match.
Message Event	Event that happens when a message is sent.	EVENT (conflated with the generic Event concept)	Partial	No distinct Message-Event subtype exists apart from an ordinary EVENT.
Organization	A collection of agents with specified roles co-operating towards a system goal.	Director's role_agent(Role,Agent) registry / running Plot instance	Full	Direct match.

Continued on next page

Table A.2 – continued from previous page

FAML Concept	FAML Definition	Regia Mapping	Support	Reasoning
Plan	An organized collection of actions that can be executed to pursue a particular agent goal.	Compiled AgentSpeak plan (from a WHEN block)	Full	Direct match.
Resource	Something that has a name, may have reasonable representations and that can be acquired, shared or produced.	(none)	None	Regia has no resource concept.
Role	Specification of a behavioural pattern expected from some agents in a given system.	ROLE / role_agent binding	Full	Direct match.
System	Final product of an agent-oriented software development project.	Running Plot (Director agent and bound Role agents)	Partial	No explicit System runtime wrapper exists beyond a single Plot instance.

Appendix B

Default Benchmark Program

The following listing reproduces the Regia program the benchmark generator of Section 7.2.1 produces for the default configuration of Section 7.2.1.3: 10 Actions, 10 Events, 5 Facts, 2 Roles, 2 Phases, and 2 Playbooks with 2 Plans each, 1 branch per Plan, and 2 statements per branch.

```
1 # =====
2 # Generated by Regia Benchmark Generator
3 # n_actions=10 n_events=10 n_facts=5
4 # n_playbooks=2 n_plans=2
5 # n_branches=1 n_stmts=2
6 # n_roles=2 n_phases=2
7 # n_subplot_breadth=0 n_subplot_depth=0
8 # seed=42
9 # =====
10
11 # == Actions ==
12 ACTION act_0.
13 ACTION act_1.
14 ACTION act_2.
15 ACTION act_3.
16 ACTION act_4.
17 ACTION act_5.
18 ACTION act_6.
19 ACTION act_7.
20 ACTION act_8.
21 ACTION act_9.
22
23 # == Events ==
```

```

24 EVENT evt_0.
25 EVENT evt_1.
26 EVENT evt_2.
27 EVENT evt_3.
28 EVENT evt_4.
29 EVENT evt_5.
30 EVENT evt_6.
31 EVENT evt_7.
32 EVENT evt_8.
33 EVENT evt_9.
34
35 # == Facts ==
36 FACT fct_0.
37 FACT fct_1.
38 FACT fct_2.
39 FACT fct_3.
40 FACT fct_4.
41
42 # =====
43 # Playbooks
44 # =====
45
46 PLAYBOOK Playbook0:
47   WHEN evt_0:
48     IF fct_0:
49       DO act_0.
50       DO act_1.
51     ELSE:
52       DO act_0.
53   WHEN evt_1:
54     IF fct_1:
55       DO act_2.
56       DO act_3.
57     ELSE:
58       DO act_1.
59
60 PLAYBOOK Playbook1:
61   WHEN evt_2:
62     IF fct_2:
63       DO act_4.
64       DO act_5.
65     ELSE:
66       DO act_2.
67   WHEN evt_3:
68     IF fct_3:
69       DO act_6.
70       DO act_7.

```

```

71         ELSE:
72             DO act_3.
73
74         # =====
75         # Main Plot
76         # =====
77
78     PLOT MainPlot.
79
80     PHASE phase_0 INITIAL.
81     PHASE phase_1.
82
83     ROLE Role0.
84     ROLE Role1.
85
86     DURING PLOT:
87         WHEN evt_0:
88             END PLOT.
89
90     DURING phase_0:
91         ON ENTER:
92             ASSIGN Playbook0 TO Role0.
93             WORLD DO act_0.
94             ASSIGN Playbook1 TO Role1.
95             WORLD DO act_1.
96
97         ON EXIT:
98             UNASSIGN Playbook0 FROM Role0.
99             UNASSIGN Playbook1 FROM Role1.
100
101         WHEN evt_1:
102             TRANSITION TO phase_1.
103
104     DURING phase_1:
105         ON ENTER:
106             ASSIGN Playbook0 TO Role0.
107             WORLD DO act_0.
108             ASSIGN Playbook1 TO Role1.
109             WORLD DO act_1.
110
111         ON EXIT:
112             UNASSIGN Playbook0 FROM Role0.
113             UNASSIGN Playbook1 FROM Role1.
114
115         WHEN evt_2:
116             END PLOT.

```

Bibliography

- [1] ABDALLA, R., AND MISHRA, A. Agent-oriented software engineering methodologies: Analysis and future directions. *Complexity* 2021 (Jan. 2021).
- [2] ALACA, O. F., TEZEL, B. T., CHALLENGER, M., GOULÃO, M., AMARAL, V., AND KARDAS, G. AgentDSM-Eval: A framework for the evaluation of domain-specific modeling languages for multi-agent systems. *Computer Standards & Interfaces* 76 (June 2021), 103513.
- [3] ANGILICA, D., IANNI, G., AND PACENZA, F. Declarative AI design in Unity using answer set programming. In *2022 IEEE Conference on Games (CoG)* (2022), IEEE Press, pp. 417–424.
- [4] ARTICY SOFTWARE. articy:draft x — narrative design tool for interactive projects. <https://www.articy.com/en/>, 2026. Accessed 3 September 2026.
- [5] BETTS, J., MUNTEAN, W., KIM, D., AND KAO, S.-C. Evaluating different scoring methods for multiple response items providing partial credit. *Educational and Psychological Measurement* 82, 1 (2022), 151–176.
- [6] BEYDOUN, G., LOW, G., HENDERSON-SELLERS, B., MOURATIDIS, H., GOMEZ-SANZ, J. J., PAVON, J., AND GONZALEZ-PEREZ, C. FAML: A generic metamodel for MAS development. *IEEE Transactions on Software Engineering* 35, 6 (Nov. 2009), 841–863.
- [7] BOISSIER, O., BORDINI, R. H., HÜBNER, J. F., RICCI, A., AND SANTI, A. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming* 78, 6 (2013), 747–761.
- [8] BORDINI, R. H., HÜBNER, J. F., AND WOOLDRIDGE, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. John Wiley & Sons, Chichester, UK, 2007.
- [9] BRATMAN, M. E. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

- [10] BRESCIANI, P., PERINI, A., GIORGINI, P., GIUNCHIGLIA, F., AND MYLOPOULOS, J. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems* 8, 3 (May 2004), 203–236.
- [11] BREWKA, G., EITER, T., AND TRUSZCZYŃSKI, M. Answer set programming at a glance. *Commun. ACM* 54, 12 (Dec. 2011), 92–103.
- [12] DAGRE CONTRIBUTORS. Dagre: Directed graph layout for JavaScript. <https://github.com/dagrejs/dagre>, 2026. Accessed 29 September 2026.
- [13] DELOZIER, C., AND SHEY, J. Using visual programming games to study novice programmers. *International Journal of Serious Games* 10 (06 2023), 115–136.
- [14] EFRON, B., AND TIBSHIRANI, R. J. *An Introduction to the Bootstrap*. Chapman & Hall, New York, 1993.
- [15] EROL, K., HENDLER, J., AND NAU, D. S. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)* (1994), vol. 2, AAAI Press, pp. 1123–1128.
- [16] FERRANDO, A., GATTI, A., AND MASCARDI, V. Integrating virtual reality, chatbots, and BDI agents: VEsNA goes fast! In *Agents and Multi-Agent Systems Development: Platforms, Toolkits, Technologies*, R. Collier, V. Mascardi, and A. Ricci, Eds. Springer Nature Switzerland, Cham, 2026, pp. 289–322.
- [17] FURTADO, A. W. B., AND SANTOS, A. L. M. Using domain-specific modeling towards computer games development industrialization. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM’06)* (Portland, OR, 2006).
- [18] GATTI, A. Raise BDI agents, first steps. In *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems* (Richland, SC, 2026), AAMAS ’26, International Foundation for Autonomous Agents and Multiagent Systems, pp. 4002–4004.
- [19] GATTI, A., CASALE, F., MASCARDI, V., STUCCHI, A., AND FERRANDO, A. VEsNA-Pro: Exploiting BDI agents with propensities for emergent narrative. In *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems* (Richland, SC, 2026), AAMAS ’26, International Foundation for Autonomous Agents and Multiagent Systems, pp. 967–976.
- [20] GATTI, A., AND MASCARDI, V. Towards VEsNA, a framework for managing virtual environments via natural language agents. *Electronic Proceedings in Theoretical Computer Science* 362 (07 2022), 65–80.

- [21] GATTI, A., AND MASCARDI, V. VEsNA, a framework for virtual environments via natural language agents and its application to factory automation. *Robotics 12* (03 2023), 46.
- [22] GEORGEFF, M. P., PELL, B., POLLACK, M. E., TAMBE, M., AND WOOLDRIDGE, M. The Belief-Desire-Intention model of agency. In *Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL '98)* (Berlin, Heidelberg, 1999), vol. 1555 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 1–10.
- [23] GODOT ENGINE CONTRIBUTORS. Godot Engine. <https://godotengine.org>, 2026. Accessed 18 August 2026.
- [24] HUTONG GAMES. PlayMaker — visual scripting for Unity. <https://hutonggames.com/>, 2026. Accessed 4 September 2026.
- [25] INKLE STUDIOS. Ink: A scripting language for writing interactive narrative. <https://www.inklestudios.com/ink/>, 2026. Accessed 20 August 2026.
- [26] IOVINO, M., SCUJKINS, E., STYRUD, J., ÖGREN, P., AND SMITH, C. A survey of behavior trees in robotics and AI. *Robotics and Autonomous Systems 154* (2022), 104096.
- [27] ISLA, D. Handling complexity in the Halo 2 AI. Game Developers Conference (GDC 2005). Proceedings published by Gamasutra, <https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai>, 2005.
- [28] JENNINGS, N. R. On agent-based software engineering. *Artificial Intelligence 117*, 2 (Mar. 2000), 277–296.
- [29] JULIANI, A., BERGES, V.-P., VCKAY, E., GAO, Y., HENRY, H., MATTAR, M., AND LANGE, D. Unity: A general platform for intelligent agents. arXiv preprint arXiv:1809.02627, 2018.
- [30] KAHRAMAN, G., AND BILGEN, S. A framework for qualitative assessment of domain-specific languages. *Softw. Syst. Model. 14*, 4 (Oct. 2015), 1505 – 1526.
- [31] KARDAS, G., TEZEL, B. T., AND CHALLENGER, M. Domain-specific modelling language for Belief-Desire-Intention software agents. *IET Software 12*, 4 (Aug. 2018), 356–364.
- [32] KEDALO, A., ZYKOV, A., ASLAM, H., AND MAZZARA, M. Comparing behaviour tree and hierarchical task network planning methods for their impact on player experience. In *2023 IEEE Symposium Series on Computational Intelligence (SSCI)* (2023), IEEE, pp. 135–139.

- [33] LARK CONTRIBUTORS. Lark: A parsing toolkit for Python. <https://github.com/lark-parser/lark>, 2026. Accessed 29 September 2026.
- [34] LIKERT, R. A technique for the measurement of attitudes. *Archives of Psychology* 22, 140 (1932), 1–55.
- [35] MARIANI, S., AND OMICINI, A. Game engines to model MAS: A research roadmap. In *Proceedings of the 17th Workshop "From Objects to Agents" (WOA 2016)* (Catania, Italy, 2016), vol. 1664 of *CEUR Workshop Proceedings*, CEUR-WS, pp. 106–111.
- [36] MATANO, F. Development of a custom C++ interpreter for VEsNA-ProC. Master’s thesis, University of Genoa, in preparation, 2026.
- [37] MATEAS, M., AND STERN, A. A behavior language for story-based believable agents. *IEEE Intelligent Systems* 17, 4 (July 2002), 39 – 47.
- [38] MATEAS, M., AND STERN, A. Structuring content in the Façade interactive drama architecture. In *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (2005), AIIDE’05, AAAI Press, pp. 93–98.
- [39] MEDLER, B., AND MAGERKO, B. Scribe: a tool for authoring event driven interactive drama. In *Proceedings of the Third International Conference on Technologies for Interactive Digital Storytelling and Entertainment* (Berlin, Heidelberg, 2006), TIDSE’06, Springer-Verlag, pp. 139–150.
- [40] MERNIK, M., HEERING, J., AND SLOANE, A. M. When and how to develop domain-specific languages. *ACM Computing Surveys* 37, 4 (Dec. 2005), 316–344.
- [41] META PLATFORMS. React: The library for web and native user interfaces. <https://react.dev/>, 2026. Accessed 29 September 2026.
- [42] MICROSOFT. Monaco Editor. <https://microsoft.github.io/monaco-editor/>, 2026. Accessed 29 September 2026.
- [43] MILLINGTON, I. *AI for Games*, 3 ed. CRC Press, Boca Raton, FL, 2019.
- [44] ORKIN, J. Three states and a plan: The A.I. of F.E.A.R. In *Game Developers Conference (GDC 2006)* (San Jose, CA, 2006).
- [45] PADGHAM, L., AND WINIKOFF, M. Prometheus: a methodology for developing intelligent agents. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1* (New York, NY, USA, 2002), AA-MAS ’02, Association for Computing Machinery, pp. 37–38.
- [46] PALLETS. Click: Command line interface creation kit. <https://click.palletsprojects.com/>, 2026. Accessed 29 September 2026.

- [47] PASCUCCI, L. Synapsis - middleware per l'integrazione di game engine e sistemi multi-agente. Master's thesis, Alma Mater Studiorum - Università di Bologna, Cesena, Italy, October 2019. Corso di Studio in Ingegneria e Scienze Informatiche [LM-DM270]. Advisors: Andrea Omicini, Stefano Mariani.
- [48] PLAY FRAMEWORK CONTRIBUTORS. Play Framework. <https://www.playframework.com/>, 2026. Accessed 29 September 2026.
- [49] POLI, N. Game engines and MAS: BDI & artifacts in Unity. Master's thesis, Alma Mater Studiorum - Università di Bologna, Cesena, Italy, March 2018. Corso di Studio in Ingegneria e Scienze Informatiche [LM-DM270]. Advisors: Andrea Omicini, Stefano Mariani.
- [50] PYTEST-DEV. pytest: Helps you write better programs. <https://docs.pytest.org/>, 2026. Accessed 29 September 2026.
- [51] RAMÍREZ, S. FastAPI. <https://fastapi.tiangolo.com/>, 2026. Accessed 29 September 2026.
- [52] RAO, A. S. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW '96)* (Berlin, Heidelberg, 1996), vol. 1038 of *Lecture Notes in Computer Science*, Springer, pp. 42–55.
- [53] RIEDL, M. O., AND BULITKO, V. Interactive narrative: An intelligent systems approach. *AI Magazine* 34, 1 (Mar. 2013), 67–77.
- [54] SCHMIDT, D., RAUPACH, T., WIEGAND, A., HERRMANN, M., AND KANZOW, P. Relation between examinees' true knowledge and examination scores: Systematic review and exemplary calculations on Pick-N items. *Educational Research Review* 37 (2022), 100483.
- [55] SECRET LAB. Yarn Spinner: The friendly tool for writing interactive dialogue. <https://yarnspinner.dev/>, 2026. Accessed 18 August 2026.
- [56] SPEARMAN, C. The proof and measurement of association between two things. *The American Journal of Psychology* 15, 1 (1904), 72–101.
- [57] UNTOLD GAMES. Untold Games: Video games with deep storytelling and a strong emotional impact. <https://untoldgames.com/>, 2026. Untold Games s.r.l., Genoa, Italy. Accessed 29 September 2026.
- [58] VAN DEURSEN, A., KLINT, P., AND VISSER, J. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices* 35 (06 2000), 26–36.

- [59] VASSALLO, E. Regia: A domain-specific language for authoring BDI agent behaviour in games. <https://github.com/inkeaton/regia/tree/1.0>, 2026. Source code, benchmarks, and case study. Accessed 17 September 2026.
- [60] WOHLIN, C., RUNESON, P., HÖST, M., OHLSSON, M. C., REGNELL, B., AND WESSLÉN, A. *Experimentation in Software Engineering*. Springer, Berlin, Heidelberg, 2012.
- [61] WOOLDRIDGE, M., JENNINGS, N. R., AND KINNY, D. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems* 3, 3 (Sept. 2000), 285–312.
- [62] XYFLOW. React Flow. <https://reactflow.dev/>, 2026. Accessed 29 September 2026.
- [63] YANNAKAKIS, G. N., AND TOGELIUS, J. *Artificial Intelligence and Games*, 2 ed. Springer Nature, 2025. <https://gameaibook.org>.
- [64] YODEN, W. J. Index for rating diagnostic tests. *Cancer* 3, 1 (1950), 32–35.
- [65] ZHU, M., AND WANG, A. I. Model-driven game development: A literature review. *ACM Computing Surveys* 52, 6 (Nov. 2019).