A Unified Representation for Dialogue and Action in Computer Games: Bridging the Gap Between Talkers and Fighters

by

Philip Hanson

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2010

APPROVED:

Professor Charles Rich, Major Thesis Advisor

Professor Michael Gennert, Head of Department

Abstract

Most computer game characters are either “talkers,” i.e., they engage in dialogue with the player, or “fighters,” i.e., they engage in actions against or with the player, and that may affect the virtual world. The reason for this dichotomy is a corresponding gap in the underlying development technologies used for each kind of character. Using concepts from task modeling and computational linguistics, we have developed a new kind of character-authoring technology which bridges this gap, thereby making it possible to create richer and more interesting characters for computer games.

Acknowledgements

I would like to thank my advisor, Prof. Charles Rich, for his dedication, attention to detail, and professionalism in preparing this thesis, as well as the Computer Science department and faculty for their support during my graduate studies.

This thesis would not have been possible without the encouragement and patient endurance of my family. I would especially like to express gratitude to my parents, John and Carolyn Hanson, for their wisdom and love during this extended process. Thank you for providing a model I can aspire to emulate.

Finally, I would like to thank the network of friends, colleagues, and mentors who provided inspiration, insight, and stimulating conversation to keep me going.

Contents

List of Figures

Chapter 1

Motivation

In most computer games, non-player characters (agents for short) are usually either "talkers" or "fighters." Talkers are the merchants, bar tenders and extras of a virtual world. They are mainly for the player to converse with—they do not modify the game state in important ways. Fighters are the monsters, companions and villains the player encounters. Their typical capabilities are the opposite of talkers: depending on whether they are friends or foes, they help or hinder the player by opening and closing doors, destroying objects, and attacking other game entities, usually with no words exchanged outside of the occasional taunt thrown in battle.

The result of this situation is that the talker and fighter character types are often one-dimensional. As with books and movies, it is less stimulating intellectually and less interesting in gameplay terms to interact with such limited characters. Agents in games have thus been limited to these narrow ranges of interaction, leaving a missed opportunity to build "integrated" characters, i.e., characters that break the dichotomy between talking and fighting.

This dichotomy between types of interactions arises because the technologies used to handle each type of interaction are disjoint. Data describing an agent's intended behavior is typically stored in a format associated with a given technology and is not used by other parts of the system. Due to the amount of effort involved in integrating the two "technology silos" of talking and fighting, most game developers refrain from including more than a handful of integrated characters—the most common example being sidekick characters. We believe that if character-building technologies were integrated, developers could include more characters with depth in computer games.

The remainder of this thesis describes our work in developing a tool called Tizona for building integrated characters and interactions in games. Following sections in this chapter provide background on current technology and the techniques we select to improve upon it. Chapter 2 introduces a game we have developed using this tool, from which we will draw examples throughout the thesis. Chapter 3 further describes the foundations of task modeling and collaborative discourse theory upon which we build and describes the game development tool we have produced. We follow this with an evaluation of Tizona's effectiveness in Chapter 4 and a summary of related work in Chapter 5. Chapter 6 states conclusions and describes future work.

1.1 Talking Technologies

Presently, the dominant technology for building talking agents is a dialogue tree [5]. Figure 1.1 shows an example of a dialogue tree from the game *Icewind Dale* in typical indented textual form as typically developed in, e.g., a spreadsheet. Figure 1.2 shows the same dialogue tree graphically. Nodes within the tree represent player or agent utterances. At each level in the tree we denote candidate player utterances with bracketed numbers before the utterances. Lines without bracketed numbers are agent utterances. Conversations between the player and an agent begin at the root of the agent's current dialogue tree. When it is the player's turn to speak, a menu of the current utterance choices is presented to the player, where each choice is a child node of the current dialogue node. A complete conversation can thus be thought of as a path through the dialogue tree from the root to a leaf.

Dialogue trees allow developers to implement conversations between the player and agents without performing natural language processing on player input [1]. However, dialogue trees are well known to suffer from a form of combinatorial explosion due to the number of possible paths. If each agent utterance has several possible player responses, the number of paths increases dramatically with the length of the conversation, and each path must be authored by the designer. This explosion cannot be totally avoided, but it may be mitigated somewhat by the automatic

```
Greetings, traveler! The name's Quimby, and I welcome you to my inn!
   [1] Well met, Quimby. You seem rather... enthusiastic.
      Ah! Enthusiastic! Yes! It is rare that I get any travelers at this time of yea...
         [1] Er... okay. How about a room, Quimby?
         [2] Indeed. Have you heard any interesting news lately, Quimby?
            Actually, yes. The whole town's been talking about the upcoming expediti...
               [1] Really? Tell me more about this expedition.
                  Well, Hrothgar no doubt will be leading the expedition. I've heard...
                     [1] Thanks, Quimby. I must be going. Farewell.
               [2] Thanks for the information, Quimby. I must be going. Farewell    ...
         [3] Do you get ANY travelers at this time of year?
            We do have one! Ha-HA! Yes, his name is Erevain something or another. He...
               [1] Did he bring any interesting news with him?
                  Actually, yes. The whole town's been talking about the upcoming ex...
                     ...
              [5] Interesting. Farewell, Quimby.
         [4] Hmm. Thanks, Quimby, but I must be going. Farewell.
                 .
                 .
                 .
```

Figure 1.1: Sample dialogue tree from *Icewind Dale*

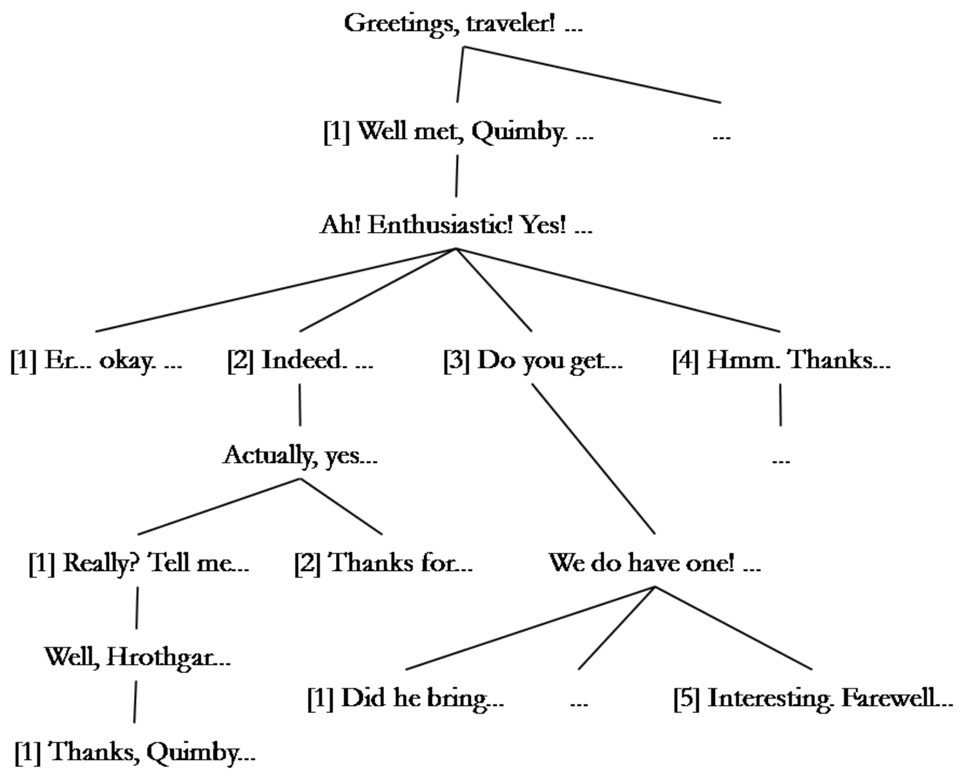


Figure 1.2: Diagram of dialogue tree from Figure 1.1

dialogue generation techniques described in Section 3.4.2.

Because fighting and talking technologies are separated, dialogue trees typically do not have side effects: regardless of which response the player selects, the state of objects in the world is usually not affected. Furthermore, all the utterances in the tree are simply strings—they lack even the most basic semantics, such as whether an utterance is a question. If conversations are to have an effect on the game world state, developers must explicitly specify these effects for each utterance and implement them in the game code. This disconnect between the conversation model and the world model also goes in the other direction, since dialogue trees are also typically static, i.e., they are pre-authored and do not change to fit the player's actions. Developers often resort to using ad-hoc hidden variables as flags to disable dialogue options and divert the conversation based on previous player actions.

1.2 Fighting Technologies

The technology that drives fighting characters is more general than just fighting: the actions performed need not be hostile. We therefore generalize our terminology, from "fighters" to "doers," meaning agents that can change the state of the world. The typical technology for implementing doers is some form of behavioral AI, such as state machines or behavior trees. We will focus on behavior trees [8], because they are becoming increasingly popular, and because they are the most similar to the talking technology of dialogue trees.

Behavior trees are essentially hierarchical task networks [9]. Using behavior trees, designers can define goals and actions for agents to pursue autonomously or as reactions to game events. These actions form a coherent model leading toward an intended goal. The process of creating a hierarchical task network is sometimes called *task modeling*, a process we also adopt and describe in greater detail in Section 1.4.1.

Behavior trees are self-describing; they are a declarative specification of agent intentions, actions, and the relationship between these elements. Since behavior trees are self-describing tree structures, they are easier to modify and less prone to accidental destruction of behaviors than state machines. But even with these advantages over previous technologies, behavior trees have typically been used only to describe actions rather than dialogue.[1]

1.3 The Problem with Modal Systems

To create agents that integrate the disjoint technologies for talking and doing discussed above, developers today typically add *more* technology to the mix. A typical approach is called a "modal interface," wherein interacting with talking agents and interacting with doing agents are performed in separate playing modes [5]. When a modal approach is used, each integrated agent must, in effect, be implemented twice: once to define their talking behavior, typically using dialogue trees, and again to define their fighting behavior, e.g., using behavior trees. In order to connect the two representations, developers insert ad hoc hidden variables to coordinate them. The process is labor-intensive and error-prone, and as a result, few integrated agents are created.

1.4 Solution Approach

Our work integrates talker and doer technologies (specifically dialogue trees and behavior trees) into a single unified representation to manage an agent's dialogue, action, and world state information. Additionally, we have developed an agent authoring tool, called Tizona, based on this unified representation. Games implemented using Tizona allow players to freely engage in both actions and dialogue with non-player characters without using a modal interface.

In order to bring talker and doer technologies together, we can either move actions into the realm of dialogue (as in text-based adventure games like *Zork*) or move dialogue into the realm of action. The majority of people do not find screenfuls of text very entertaining, so the former approach is not useful for mainstream games. To move dialogue into the realm of action we *represent utterances as a type of action.* We thus "bridge the gap" by representing talking as a kind of doing.

To create a unified representation for dialogue and action (and accompanying tools), we draw from the fields of task modeling and computational linguistics. Specifically, we draw from a branch

[1]A situation which we are about to remedy!

of computational linguistics called collaborative discourse theory, which takes both utterances and actions into account in modeling dialogue [19]. The following sections provide a brief overview of these foundations.

1.4.1 Task Modeling

Task modeling—the process of developing a model of possible player and agent goals and actions in a particular domain—is a well-known technique in both player interface design and artificial intelligence [16]. In task modeling, tasks are represented as nodes of a directed graph (which we will sometimes informally call a "tree," ignoring the possibility of "joins"). Primitive tasks, called *actions*, can be accomplished directly. For example, an action might be to push a button or pull a lever. Composite tasks, called *goals*, are higher-level constructs that are accomplished by decomposing them into their component actions. Goals may have multiple alternative methods by which they can be accomplished, called *decompositions*. We call child subgoals and actions the *steps* of a decomposition. Actions, goals, and decompositions may have conditions (expressed either in code or in a logical language) limiting the circumstances in which they can be used. An *utterance* is considered to be a type of action which results in the agent having said something.

1.4.2 Collaborative Discourse Theory

Collaborative discourse theory provides tools to model human interactions based on utterances and actions. The phrase "collaborative discourse" reflects the field's focus on the communication between two people working together to achieve a goal. According to this theory, in order to collaborate successfully, participants must hold "mutual beliefs about the goals and actions to be performed and the capabilities, intentions, and commitments" of their collaborators [17]. We represent these mutual beliefs (by way of task modeling) with a *plan tree* containing goals and actions. Additionally, collaborative discourse theory states that attention in human conversations follows the model of a stack (called the *focus stack*), where new conversation topics or sub-topics are pushed onto a stack while under discussion and later popped when the conversation returns to the previous topic.

```
[Achieve CrossRiver] -succeeded
   Player says "We need to get to the other side."
   [Get to the first island by making a bridge] -succeeded
      [Player says let's get to the first island somehow] -succeeded
         Sidekick says "How do you want to get to the first island?"
      Player push an ice block into the water. -succeeded
      [Player walk to the first island] -succeeded
         Sidekick says "Please walk to the first island."
   [Get to the second island using the rope] -succeeded
      [Player says let's get to the second island somehow] -succeeded
         Sidekick says "How do you want to get to the second island?"
      Player says "It's too far for me to swim. There's a rope, though..."
      [Sidekick swim to the second island] -accepted -done
         [Player says please swim to the second island] -done
            Sidekick says "Well, I can make it across. I'll send you a postcard
from the other side."
            Player says "Ok, in you go!"
         Sidekick swim to the second island.
      Sidekick throw the rope.
      [Player grab the rope] -done
         Sidekick says "Please grab the rope."
         Player grab the rope.
   [Get to the far side by swimming] -succeeded
      [Player says let's get to the far side somehow] -succeeded
         Sidekick says "How do you want to get to the far side?"
```

Figure 1.3: Example interaction history

Figure 1.3 is presented as a preview of our solution. We can show the natural hierarchical structure of human collaboration using an *interaction history* such as Figure 1.3. This listing shows the state of an interaction in progress between a player and an agent as they attempt to accomplish a goal from the first level of *Secrets of the Rime* (see Chapter 2). Notice that this interaction demonstrates the interleaving of actions and utterances: we see that the sidekick performed an action ("Sidekick push an ice block…"), and subsequently produced an utterance ("Please walk to…") requesting action from the player. We will discuss interaction histories in greater detail in Chapter 3.

Chapter 2

Technology Demo: *Secrets of the Rime*

We have produced a game demo called *Secrets of the Rime* that illustrates the benefits of our approach. The game and the Tizona tool were developed as co-evolving systems. In *Secrets*, the player and a sidekick character need to overcome a series of puzzles that must be solved collaboratively. Each puzzle has associated dialogue, but at any time the player or sidekick may perform an action that advances the puzzle. Players can interleave dialogue and action at will.

The following sections comprise a descriptive treatment of the game. This treatment is included here for reference so we can draw on *Secrets* for examples throughout the remaining chapters. Details of our completed implementation are omitted.

2.1 Overview

Secrets of the Rime is a single-player 2D adventure game with environmental puzzle-solving and mild role-playing elements. The game's story follows the player and a sidekick character, both researchers in Antarctica, who find themselves cut off from their research base and must take the long way back. Collaboration is an integral element of gameplay, with most puzzles requiring coordination of player and sidekick actions.

Secrets is composed of four levels, each containing a puzzle to be solved or task to be completed:

- **Ice Blocks**: cross a river

- **Ice Wall**: get over a wall

- **Shelter**: build a fort to take shelter in

- **Walrus Cave**: solve the riddles posed by a Sphinx-like walrus

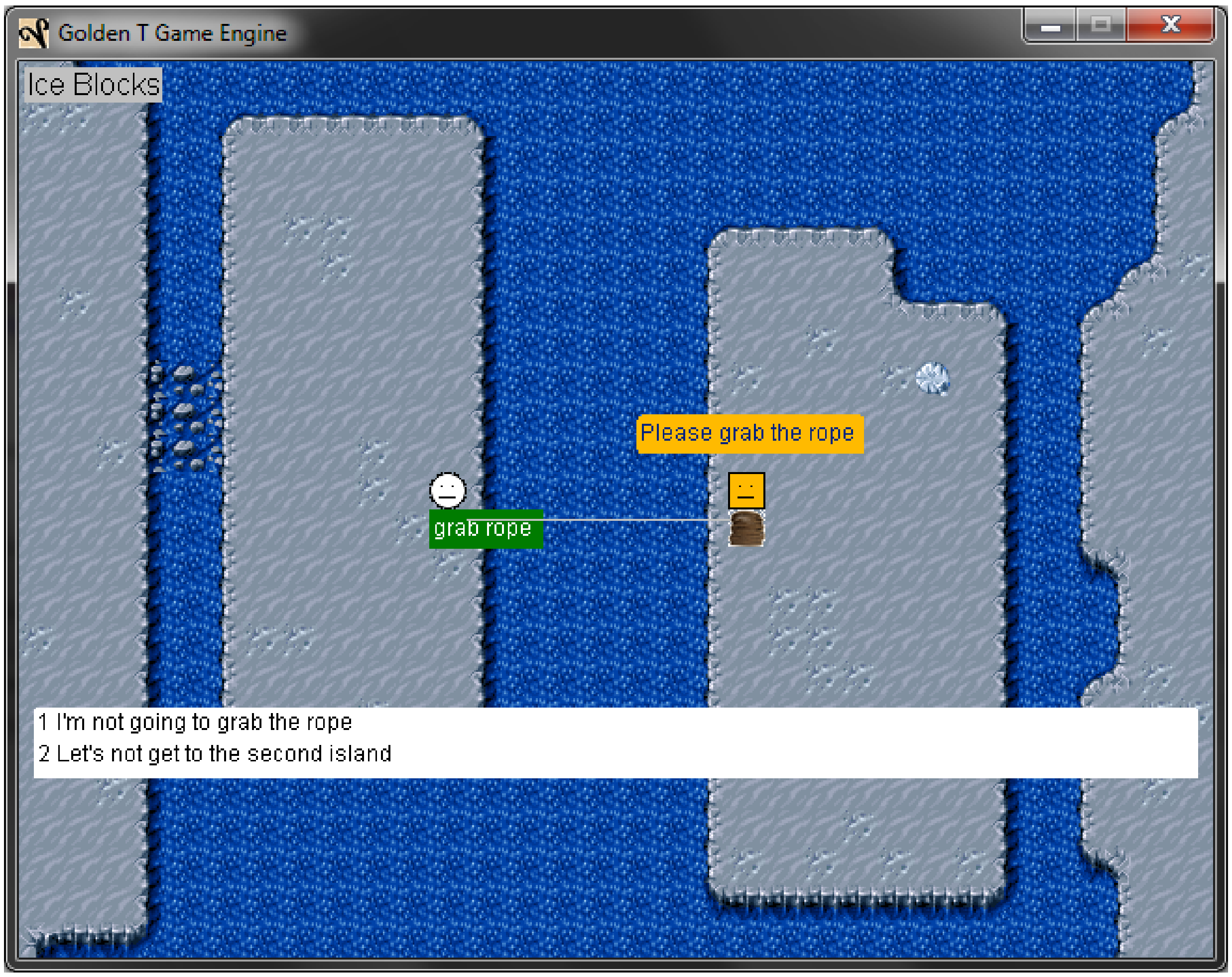
At the start of each level, the player and sidekick are placed near the left edge of the level's playing area, or "map," like that shown in Figure 2.1. They are attempting to return to their base, which is far to the right and outside the map, and various obstacles stand in their way. Thus, a level is considered completed when both parties reach the right edge of the current level's map.

Figures in this chapter, starting with Figure 2.1, show screen captures of the working demo. The graphical presentation is at a prototype level and not intended to be production quality. Thus the player is represented as a white circle and the sidekick is represented as an orange square. Player utterance choices are presented in a numbered menu at the bottom of the screen, and agent utterances are shown in a nearby comic-book-like speech bubble. The player can move around the map using the arrow keys, and can choose utterances from the menu by pressing a number on the keyboard. Objects that have associated actions display a text description of the action when the player or an agent is nearby. The player can perform actions thus advertised by an object by using the arrow keys to walk "into" the object. See the "open" object action advertisement in Figure 2.2 for an example.

In the following sections we describe each level in further detail.

2.2 Ice Blocks

Figure 2.1 shows the middle of a play session of the Ice Blocks level. The map has four areas of dry land: two river banks (on the left and right) and two islands in the middle. The rest of the map is the river itself. There are two ice blocks, one on the near river bank and one on the second island. Each block can be pushed into the water to form a bridge to the next land mass. Thus



1 I'm not going to grab the rope
2 Let's not get to the second island

Figure 2.1: Ice Blocks level

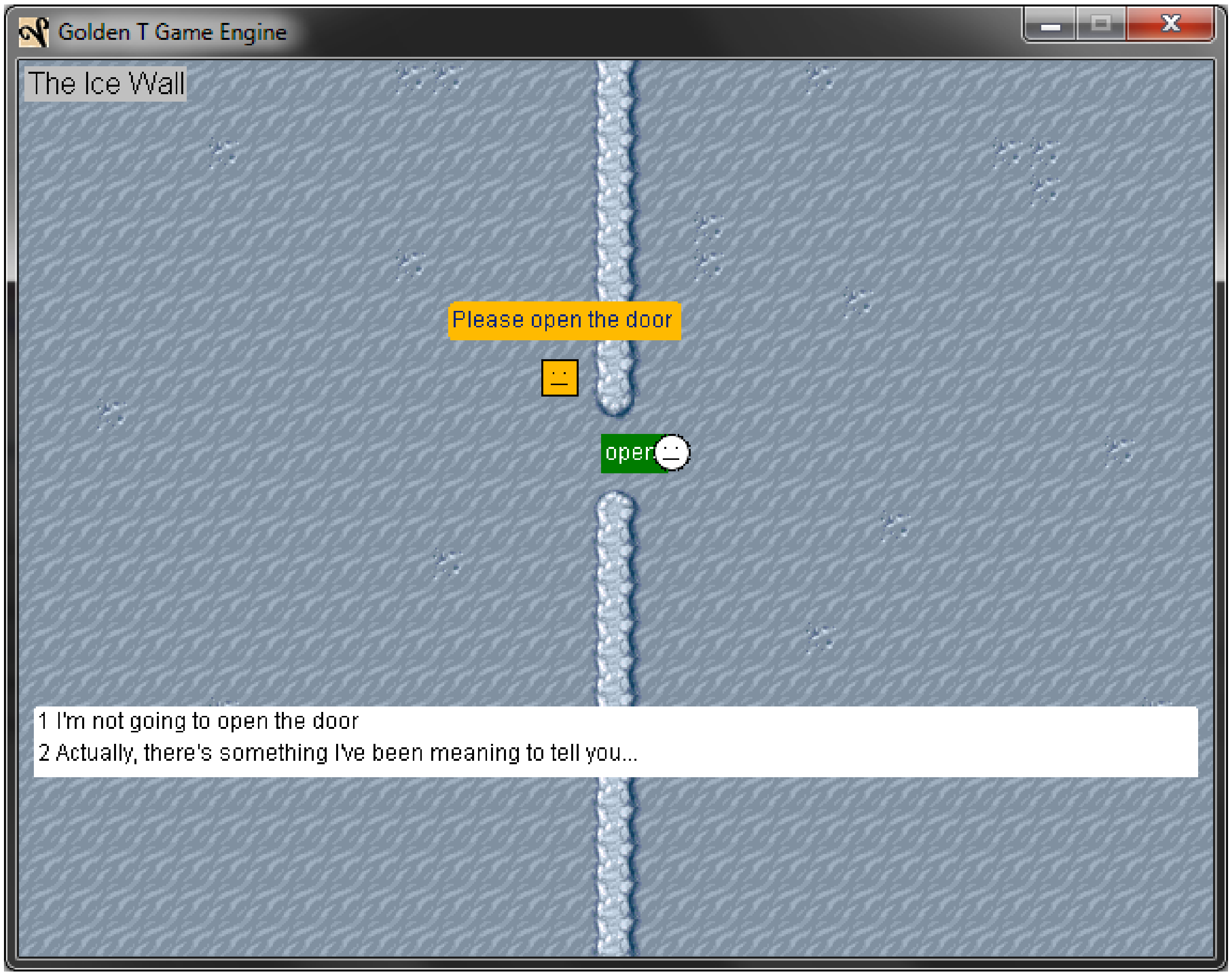
the player has several options for traversing the level. To get from the left bank to the first island (or from the second island to the right bank):

- both player and sidekick can swim,
- either player or sidekick can push the ice block into the water to create a bridge, or
- one can swim while the other uses the bridge.

All of these choices can be selected either through dialogue or by performing appropriate actions. To get from the first island to the second island, the sidekick can swim across and throw a rope to the player.

At the moment shown in Figure 2.1, the leftmost ice block has already been pushed into the water to form a bridge, which the player and sidekick have crossed. Both player and sidekick can swim, but the gap between the two islands is too wide: only the sidekick is able to swim across this gap unaided. After swimming across, the sidekick can throw a rope from the second island to the player to help him across, as shown in the figure. Other gaps are short enough for the player to swim unassisted.

2.3 Ice Wall



Please open the door

1 I'm not going to open the door
2 Actually, there's something I've been meaning to tell you...

Figure 2.2: Ice Wall level

Figure 2.2 shows one moment from a play session of the Ice Wall level. The map consists of a flat plane of snow and ice, and a wall stretched across it vertically with a door in the center. The door in the middle of the wall is locked and can only be opened from the other side. The wall is too high for one person to jump or climb over, but it is reachable by standing on another's shoulders. Cliffs on either side (off-screen) make it impossible to circumvent the wall without climbing equipment, which the characters do not have. There is a thin section of the wall that could be chipped away with other ice shards, but it would take a long time. In Figure 2.2 the sidekick has just boosted the player over the wall and is asking the player to open the door from the other side.

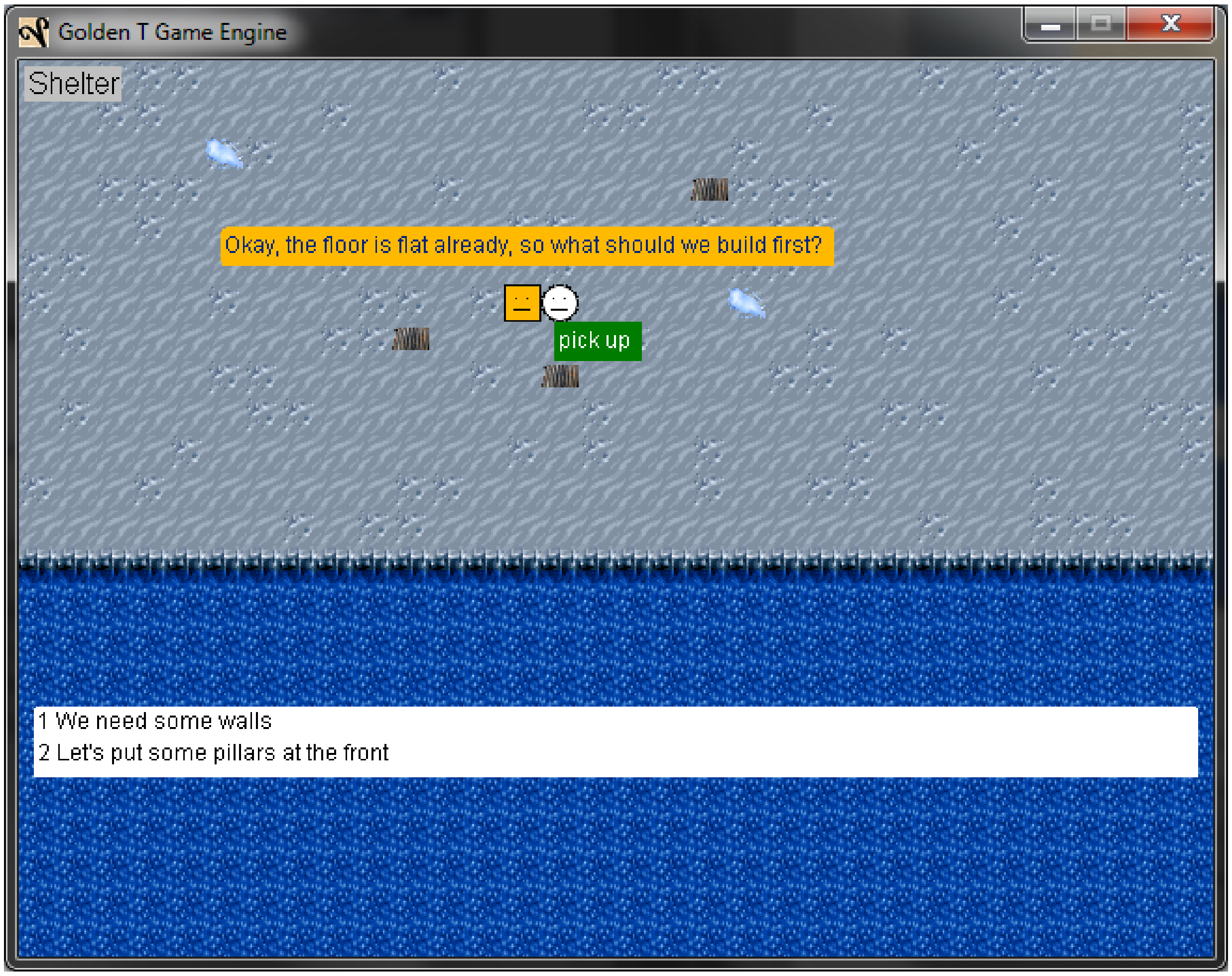
2.4 Shelter

Figure 2.3 shows the middle of a play session of the Shelter level. There is no physical obstacle to progress in this level. However, weather conditions are worsening and the characters must construct a shelter before they are caught in a storm. Either the player or the sidekick must collect ice shards and scraps from a plane wreck, place them, and then hold them steady until the other party puts a supporting piece in place. Note that these roles are *interchangeable*: any piece of the shelter can be placed by either the player or the sidekick so long as the physical conditions for placing that piece are met.

2.5 Walrus Cave

Figure 2.4 shows a moment from one play session of the Walrus Cave level, at the beginning of a three-way conversation between the player and two NPCs. (Speech bubbles are color-coded to speakers, so the speech bubble with white text on black is the walrus' dialogue.) Upon entering a large ice cave, player and sidekick are surprised to find that the exit (and their way back to base) is blocked by a talking walrus. The walrus demands that they solve a riddle, posed in verse, before he will let them through. The riddle is

> "Living threats your senses hone
>
> The frozen twin smiling is seated


Golden T Game Engine

Shelter

Okay, the floor is flat already, so what should we build first?

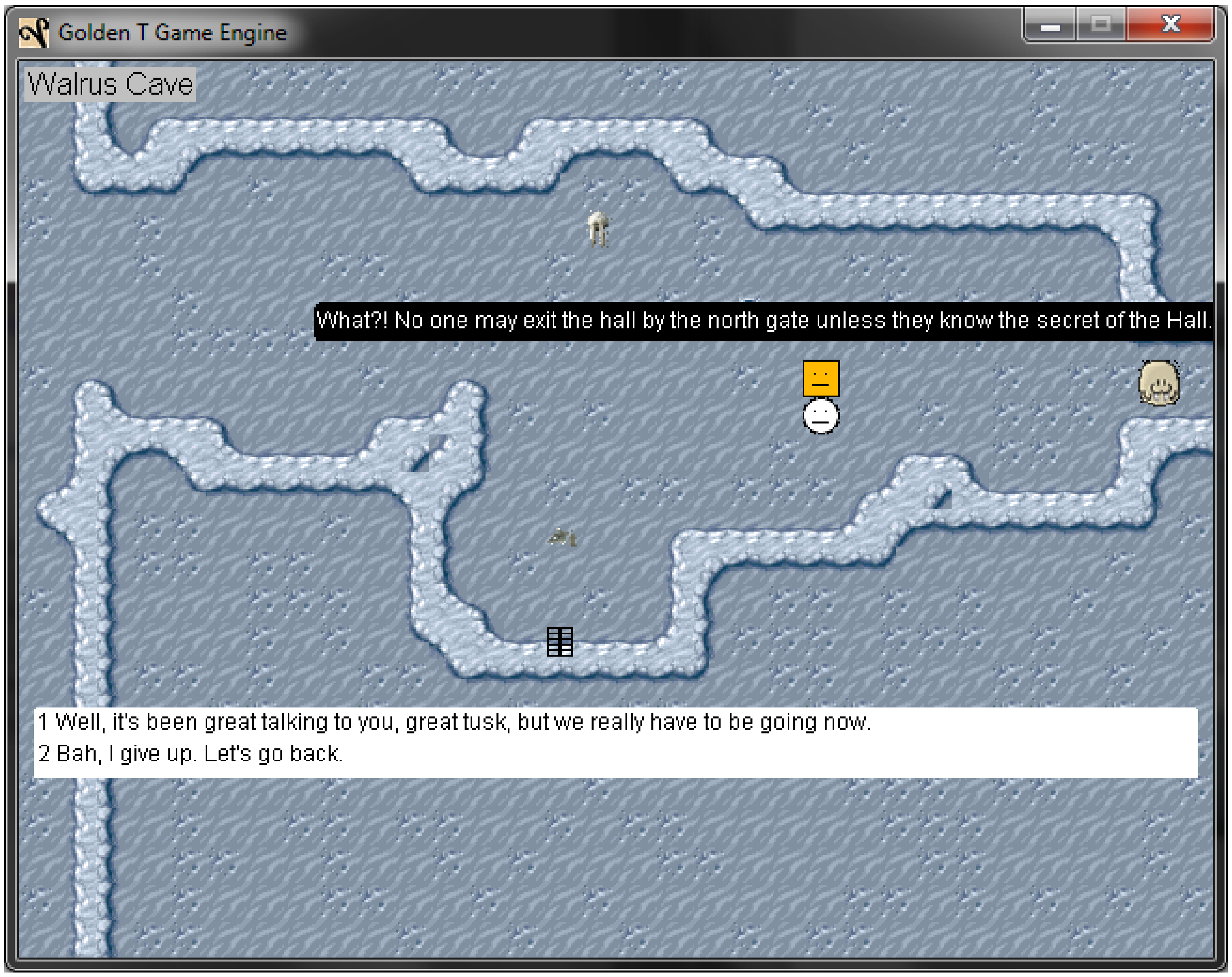
1 We need some walls

2 Let's put some pillars at the front

Figure 2.3: Shelter level

> There upon a warming throne
>
> But you must not, or be defeated
>
> Bring forth the adamantly polished
>
> Hidden when walls were breached"

and requires the player to give the correct answer of "a diamond." Once the riddle has been solved the walrus will disappear, revealing the cave's exit and the end of the game. Note that this level includes a 3-way conversation in which the player, sidekick, and walrus speak in turn.



What?! No one may exit the hall by the north gate unless they know the secret of the Hall.

1 Well, it's been great talking to you, great tusk, but we really have to be going now.
2 Bah, I give up. Let's go back.

Figure 2.4: Walrus Cave level

Chapter 3

Tizona: A Generic Non-Modal Tool for Games

Our main contribution is the development of a tool for game developers, enabling them to create games with agents that combine talking and doing without the need for a modal interface. This tool, called Tizona,[1] is used at design time to author models of agent and player behavior, and at run time to manage agent dialogue and actions.

Our goal in creating this tool was to build on existing technologies and provide a platform for future development. Tizona is built on top of two existing tools: ANSI/CEA-2018 and Disco. ANSI/CEA-2018 is a standard for representing task models in XML, and Disco is open-source software created by Prof. Charles Rich at WPI. The CETask task engine software is a reference implementation of ANSI/CEA-2018 [16], an XML task modeling standard. Disco uses task models to maintain the plan tree of an interaction and support agent behaviors. Disco is built on top of CETask and includes run time algorithms for building and using a discourse model, including a task library that defines lightweight semantics for utterance types. Disco is inspired by and based on the same collaborative discourse theory as Collagen [17]. All of these components, including Tizona, are implemented in the Java programming language.

The architecture of our work is shown in Figure 3.1. On the left is the run time component

[1]Tizona is named after the sword carried by El Cid, now housed at the Museo de Burgos in Burgos, Spain.

architecture. It shows the use of Disco as the basis for managing task models, the plan tree and focus stack, Tizona to coordinate Disco with the game world, and finally a specific game engine at the top level (though the core of Tizona is game-independent). On the right is the design time component architecture that depicts the design-time use of our unified representation (see Section 3.3.1) to create the game content necessary to use Tizona at run time. Use of this representation at design time is key to bringing together talking and doing in games. The design time architecture also includes other game content produced by the developer, such as 3D models, textures, and audio files.

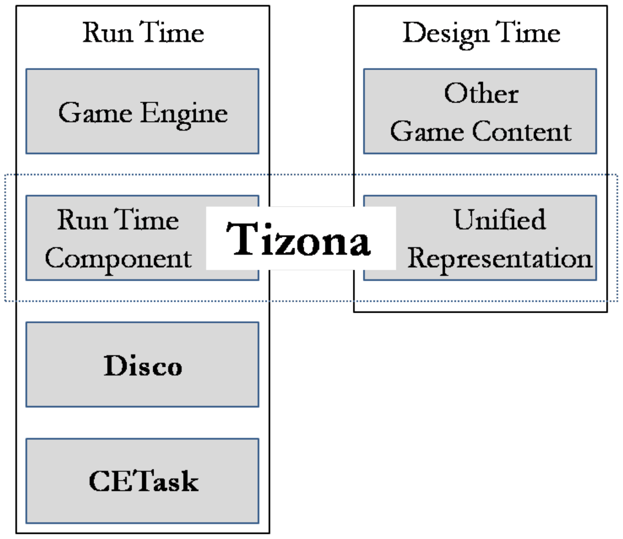


Figure 3.1: Block diagram of solution components

We have also produced an extension for Tizona that integrates with the freeware Golden T Game Engine [6]. This extension is part of the run time component of Tizona in Figure 3.1, and its primary role is to keep the game state synchronized between Tizona and Golden T. We describe this run time extension further in Section 3.3.2.

The remaining sections of this chapter proceed as follows. Section 3.1 contains a primer on ANSI/CEA-2018, the XML notation used by Disco, which is followed in Section 3.2 by a description of the Disco engine and its capabilities. With this technical foundation in place, we move on to technical aspects of our solution in Section 3.3, including our key contributions of improved task model representation, game engine API, and multi-way conversation support. We conclude the chapter in Section 3.4 with a description of some additional benefits gained using the technology we have chosen and adapted.

3.1 ANSI/CEA-2018

The ANSI/CEA-2018 standard defines an XML notation for task models. Figure 3.2 shows an example diagram of an ANSI/CEA-2018 task model, which we will use for a brief introduction to the standard. For a full reference, see the standard [4].

When working with task models, it is often useful to draw diagrams like Figure 3.2, though such diagrams do not include all the details of the full XML version. The model represented by Figure 3.2 is drawn from the Ice Blocks level of *Secrets of the Rime* (see abbreviated XML version in Figure 3.4 and the complete version in Appendix B). Goals and actions are represented by ovals, decompositions are represented as diamonds, and utterances (a type of action) as rectangles. Actions are distinguished from goals by being at the leaves of the tree. Goals are connected to decompositions using a dashed line, while decompositions are connected to actions and utterances using a solid line. Decomposition steps are by default unordered. Ordering constraints are represented using arrows.

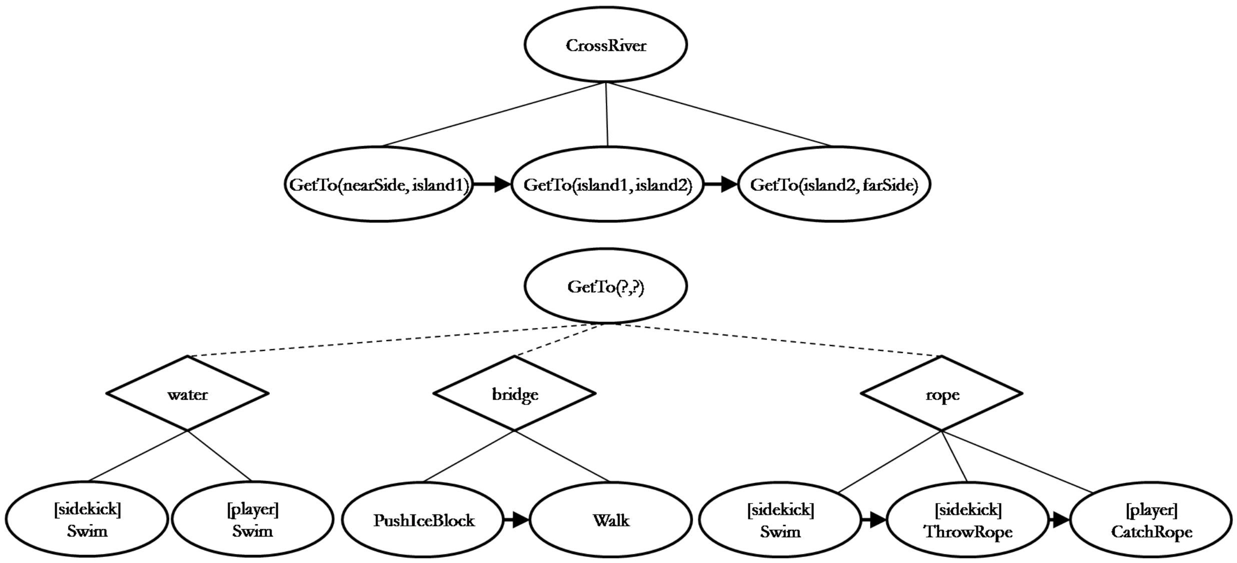


Figure 3.2: Task model for Ice Blocks level

Task models can be thought of as a type of and/or tree [22, 7]. The semantics of this example model are as follows. There is one top-level goal, CrossWater, which has three totally ordered steps (subgoals). Note that a set of steps can be thought of as an "and" operator: all actions or subgoals must be completed (subject to ordering) to complete the goal. Each subgoal of CrossWater is an instance of a parameterized goal called GetTo(?,?), where we have supplied the values (represented by "?" symbols) in the complete task model. GetTo has three alternative decompositions: at least one of 'swim,' 'bridge,' or 'rope' needs to be accomplished in order to complete GetTo. Thus we see that decompositions act as an "or" operator. Several of the actions have restrictions on who may execute the task: we represent restrictions on the actor by placing the actor's name (e.g., player or sidekick) in brackets before the task name.

We are, in essence, using task models as a high-level declarative programming language for non-player characters in computer games. The ability to represent conditionals such as "and" and "or" within a task model and using tools like partially ordered decompositions rather than ad-hoc variables and scripts is a crucial part of making the technology practical for game development.

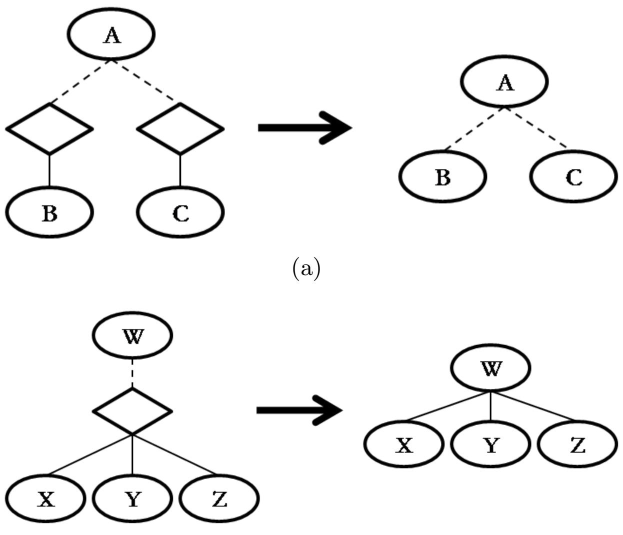


Figure 3.3: Example of diagram simplification conventions

A word on diagram convention: the left and right sides of Figure 3.3 are equivalent. When every decomposition of a goal has only a single child (whether subgoal, action, or utterance), the decompositions may be omitted and dashed lines drawn directly to the children as in (a). It is implied by the semantics of the notation that the decompositions exist even if they are not shown, as in the top half of the figure. Likewise, if only one decomposition exists for a goal, we may draw solid lines from the goal directly to the decomposition's steps as in (b). The line type implies that there is a single decomposition, because goals cannot have steps. These conventions are useful in simplifying diagrams, yet remain unambiguous. We will use these simplifications in later diagrams.

Figure 3.4 contains an abbreviated listing of the Ice Blocks task model we show in Figure 3.2. A full listing is included in Appendix B. Using Figure 3.2 for reference, we will explain the structure of the XML format. Note that in ANSI/CEA-2018, both goals and actions are declared using a 'task' element. The discourse engine determines whether a task is primitive (an action) or composite (a goal) at run time based on the presence or absence of decompositions for that type of task. Since decompositions are composed of subtasks, we use a 'subtasks' element for each decomposition.

The top-level goal, CrossWater, has a single decomposition with three steps, each of which is an instance of GetTo. We have omitted lines that supply parameter values for brevity. The task has two input parameters, 'from' and 'to,' that represent areas (locations) in the game world. The type of a parameter can be any valid JavaScript type or Java class or interface; the 'to' and 'from' parameters are declared here as Java types from the game using the Packages syntax afforded by LiveConnect [14]. GetTo also has a postcondition, a JavaScript expression whose value is supposed to evaluate to true upon goal success. There can also be 'precondition' elements that determine whether a task is eligible for execution.

The definition of GetTo has three alternative decompositions, named 'water,' 'bridge,' and 'rope.' According to the model, then, one can accomplish GetTo by performing all the steps of one of these decompositions. The 'applicable' element in the 'water' decomposition is similar to a precondition in nature—it is a JavaScript expression used to determine whether a decomposition can be applied to the current situation. The specific precondition for 'water' restricts execution to instances not involving the second island as a destination, because the gap is too big. This expression uses the 'world' variable to access the state of the game world, which we discuss further

```
<taskModel about="urn:cetask.wpi.edu:models:secrets:IceBlocks"
  xmlns="http://www.cs.wpi.edu/~rich/cetask/cea-2018-ext"
  xmlns:disco="urn:disco.wpi.edu:Disco">

  <!-- Top-level goal -->
  <task id="CrossRiver">
    <subtasks id="cross">
      <step name="goIsland1" task="GetTo"/>
      <step name="goIsland2" task="GetTo"/>
      <step name="goFarSide" task="GetTo"/>
       ...
    </subtasks>
  </task>

  <!-- Parameterized subgoal -->
  <task id="GetTo">
    <input name="from" type="Packages.edu.wpi.secrets.objects.Area"/>
    <input name="to" type="Packages.edu.wpi.secrets.objects.Area"/>
    <postcondition> ... </postcondition>

    <subtasks id="water" ordered="false">
      <step name="playerSwim" task="Swim"/>
      <step name="sidekickSwim" task="Swim"/>
      <applicable>
        $this.to != world.get("island2")
      </applicable>
      <binding slot="$sidekickSwim.from" value="$this.from"/>
      <binding slot="$sidekickSwim.to" value="$this.to"/>
      <binding slot="$sidekickSwim.external" value="false"/>
       ...
    </subtasks>
    <subtasks id="bridge">
      <step name="pushBlock" task="PushIceBlock"/>
      <step name="walk" task="Walk"/>
       ...
    </subtasks>
    <subtasks id="rope"> ... </subtasks>
  </task>

  <!-- Actions -->
  <task id="Walk"> ... </task>
  <task id="Swim"> ...  </task>
  <task id="PushIceBlock"> ... </task>
  <task id="ThrowRope"> ... </task>
  <task id="CatchRope"> ... </task>
</taskModel>
```

Figure 3.4: Example task model in ANSI/CEA-2018 XML format

in Section 3.3.2. Several steps in the 'water' decomposition have parameters that are supplied with values using a 'binding' element. One such binding supplies a value for the built-in 'external' parameter. By binding the value of 'external,' we can specify which actor can perform the action, as is shown by the bracketed names in Figure 3.2. In this instance, we restrict the execution of a Swim action to the sidekick.

Note that ANSI/CEA-2018 (and therefore Disco) only natively supports interactions with two actors, reflected in the use of a *boolean* parameter 'external'. By default, Tizona considers the player to be the external=true role and the NPC to be the external=false role. Tizona also expands the number of actors that can participate in an interaction (see Section 3.3.3).

3.2 Disco

Disco is a collaborative discourse engine. After we have authored a model using ANSI/CEA-2018, we provide it as input to Disco, which interprets and guides an interaction. At any point, Disco's internal state will reflect the structure and chronology of the interaction according to the task model.

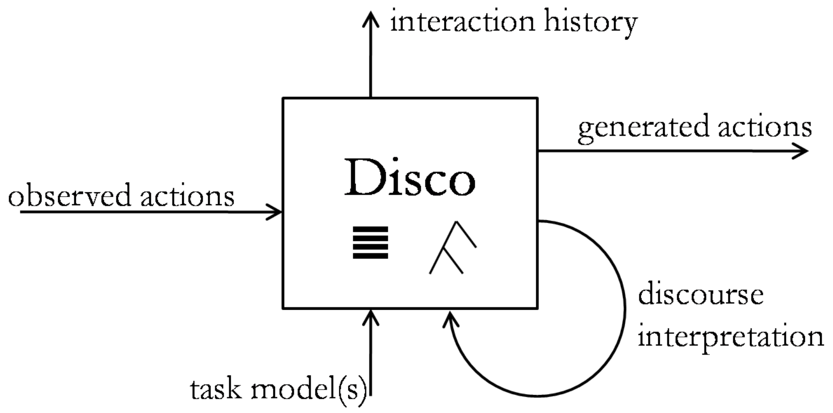


Figure 3.5: Disco inputs and outputs

Figure 3.5 illustrates Disco's external operation. We see that Disco takes as input one or more task models. Disco's internal state is indicated by two icons representing the focus stack and the plan tree specified by collaborative discourse theory. These elements represent the discourse state that is produced using elements from the task model (for details of Disco's internal operation see [16, 17]). The other input to Disco is observations of the actions that are being performed by both (all) participants in the interaction—including utterances, because we consider them to be a type of action. The engine processes each incoming action based on where it fits in the task model structure according to the action's semantics and current discourse state, and then updates the discourse state (discourse interpretation).

Disco also produces output in the form of a list of possible actions for each actor to perform, where these actions would be expected in relation to the current discourse state. Thus, Disco can suggest the next NPC action or utterance. Disco acts as an NPC control engine by generating appropriate actions for NPCs to perform, and can also generate a menu of possible player utterances (cf. Section 3.4.2). Note, however, that Disco is designed for interactions of exactly two participants, as supported by the ANSI/CEA-2018 standard. See Section 3.3.3 for our approach to generalizing to interactions with more than two participants.

Interaction Histories

```
[Achieve CrossRiver] -live
   Player says "We need to get to the other side."
   [Get to the first island by making a bridge] -live
      [Player says let's get from the near side to the first island somehow] -done
         Sidekick says "How do you want to get to the first island?"
      Player push an ice block into the water. -succeeded
      [Player walk to the first island] -live <-focus
         Sidekick says "Please walk to the first island."
   [Get to the second island]
   [Get to the far side]
```

Figure 3.6: Example interaction history

The other output of Disco shown in Figure 3.5 is the interaction history, which is a textual visualization of the current discourse state. An interaction history shows the past, present, and expected future of an interaction. Interaction histories are a useful tool when debugging task models, and they can also provide a form of post-play review for players. Figure 3.6 is an example interaction history from the Ice Blocks level at the moment when the player has pushed an ice block into the water between the near shore and the first island. Goals are denoted in Disco interaction histories by placing brackets around the corresponding goal description (note that we use Disco's facility for customizing the printing of goal descriptions, described in Section 3.4.2). Referring to the Ice Blocks model in Figure 3.2, we see that the top-level goal CrossRiver was initiated by a player utterance. The three GetTo subgoals have been added to the tree (shown at a greater indentation level), but a decomposition has only been chosen for one, so only that instance is expanded. The listing continues hierarchically through the levels of the task model until an action or utterance is reached. Below the first subgoal there is an utterance in which the sidekick asks the player to choose a decomposition, followed by an action and another utterance regarding the next step. The remaining subgoals show actions that must be completed in order to accomplish the goal. The last two lines of this interaction history show expected future goals that are not yet live.

Figure 3.6 also exhibits other elements of the discourse state: the "focus" pointer shows the location of the top of the focus stack, and various flags indicate action completion and applicability. Three tasks in the figure are marked as "live," which signifies that they are eligible for execution. Two tasks have been completed, and these are marked as "done" and "succeeded." The "done" flag denotes tasks that have been executed. For tasks that include a postcondition, the done flag is specialized into "succeeded" (as well as "failed") to indicate the result of execution. See Section 3.4.4 for further discussion of of failure modeling.

3.3 Solving the Problem of Modal Systems

As introduced in Section 1.4, we eliminate the technology gap between talking and doing by folding both technology types into a single *unified representation*. The existence of a unified representation allows us to handle both utterances and actions using the same agent control engine (Disco). Use of a single engine means that there is no longer a need to switch between "talking mode" and "acting mode," and it also means that dialogue state does not need to be synchronized between multiple engines. Instead there is a single mode, which we might call "interacting mode", where the player and all NPCs may at any time produce an utterance or perform an action.

In this section we describe an application of task modeling and collaborative discourse theory to agents in computer games. In Section 3.3.1 we describe a unified representation that extends the ANSI/CEA-2018 standard, simplifying common cases without loss of generality. In Section 3.3.2 we describe the API exposed by Tizona for integration with game engines, and we also outline Tizona's support for multi-way conversations in Section 3.3.3.

Figure 3.7 shows an example diagram of a task model that includes utterances. The figure demonstrates the key idea of interleaving actions and utterances in the same task tree. All utterances shown are instances of Disco's "Say" task class, which has a single input parameter that contains the text of the utterance. We omit the names of intermediate goals where they have no bearing on the structure or behavior of the model.

Figure 3.7 is taken from the Shelter level of *Secrets of the Rime*, where the goal of the level is to build a structure from existing materials. Unlike the Ice Blocks model (Figure 3.2), we do not introduce the objective of the level as a top-level goal; here we include preliminary dialogue between the player and sidekick which implicitly leads to the introduction of the objective as a goal. As we will show, the upper portion of the diagram is essentially a dialogue tree while the internal structure of the model makes use of decompositions and ordering to interleave actions and utterances. We will describe a more compact representation for the special case of dialogue trees in task models in Section 3.3.1.

There is one top-level goal in Figure 3.7. The player's initial dialogue option corresponds to that goal, which initiates the conversation. After the player has selected the top-level goal, the only available option is for the sidekick to produce its line of dialogue. After the sidekick's second utterance, we see from the presence of two alternative decompositions that the player will have two utterance choices. Disco will produce a list of utterances as a menu for the player containing the two player utterances that are one level below in the hierarchy. Figure 3.8 shows an example interaction history of this choice and associated utterances. By selecting one of these utterances, the player will implicitly choose which decomposition to use. (The listing in Figure 3.8 comes from Tizona's development and debugging shell, which allows developers to look at interaction histories and generate player menus without using a game engine.)

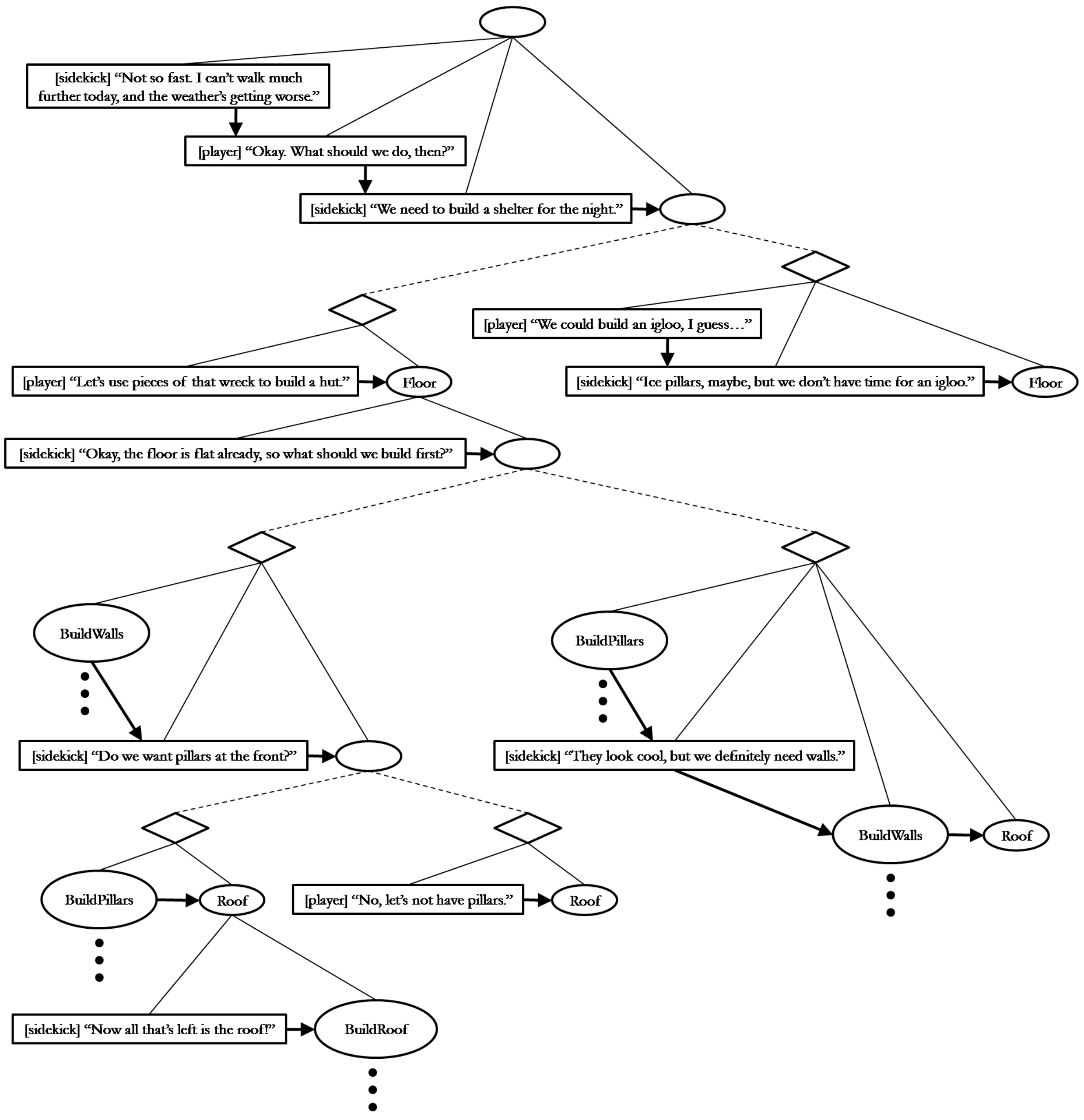

[sidekick] "Not so fast. I can't walk much further today, and the weather's getting worse."
[player] "Okay. What should we do, then?"
[sidekick] "We need to build a shelter for the night."
[player] "We could build an igloo, I guess…"
[sidekick] "Ice pillars, maybe, but we don't have time for an igloo."
Floor
[player] "Let's use pieces of that wreck to build a hut."
Floor
[sidekick] "Okay, the floor is flat already, so what should we build first?"
BuildWalls
[sidekick] "Do we want pillars at the front?"
BuildPillars
[sidekick] "They look cool, but we definitely need walls."
BuildWalls
Roof
BuildPillars
Roof
[player] "No, let's not have pillars."
Roof
[sidekick] "Now all that's left is the roof!"
BuildRoof

Figure 3.7: Partial task model for Shelter level

```
> history
[Achieve Shelter] -live
   Player says "All right, a clear path for once!"
   Sidekick says "Not so fast. I can't walk much further today, and the weather's
getting worse."
   [ ] -live
      Player says "Okay. What should we do, then?"
      [ ] -live <-focus
         Sidekick says "We need to build a shelter for the night."
         [ ] -live

> say
[1] Let's use pieces of that wreck to build a hut.
[2] We could build an igloo, I guess...
```

Figure 3.8: Interaction history demonstrating dialogue tree behavior

We begin to introduce actions into the tree after the second set of decompositions. The left subtree, beginning with the player utterance "We need some walls," requires the subgoal BuildWalls to be accomplished before continuing to the next utterance in the dialogue tree. The subgoals of BuildWalls, BuildPillars, and BuildRoof are not primitive actions—they have decompositions not shown in the figure. Interleaving utterances and actions—or even utterances and goals—is supported since the BuildWall subgoal must be expanded before executing the sidekick's "Do we want pillars at the front?" utterance. Disco guarantees that the ordering constraints are respected.

By representing the dialogue state and action conditions in the same model, we eliminate the need for hidden variables to coordinate the order of actions. We simply place actions as descendants of an appropriate decomposition, which may be chosen through dialogue.

3.3.1 *Tizona* Representation

Data files for Disco at run time must be in ANSI/CEA-2018 format, which may include utterances as described above. However, we also define a higher-level language, referred to as *Tizona* representation, as a macro facility on top of ANSI/CEA-2018. *Tizona* reduces the complexity of expressing simple, common cases—like simple dialogue trees—and the complexity of interleaving actions and utterances in ANSI/CEA-2018 by adding two new XML elements called 'do' and 'say.' (The XML schema for *Tizona* is included in Appendix C.) This higher-level representation is intended for use at design time—in order to run a model, it is translated into ANSI/CEA-2018 format using the XSL Transformation file included in Appendix D.

Using *Tizona*, we can define a dialogue tree with familiar structure similar to that of Figure 1.1. The 'say' element represents an utterance, and it requires the text of that utterance as an attribute. 'Say' elements are naturally expanded into Disco "Say" tasks. Figure 3.9 shows a partial listing of the *Tizona* format model for the Ice Wall level of *Secrets of the Rime*. This portion of the model is in essence a simple dialogue tree. Figure 3.10 is a task model diagram of the same portion of the model. The full listing is included in Appendix B. The segment reproduced here corresponds to the portion of Figure 3.10 above the Escape task, namely the initial conversation of the level.

The nesting of elements in *Tizona* has semantic implications: child elements are understood to follow the parent in the conversation, and sibling elements are understood to be alternative decompositions. Thus, sequences are represented by repeated nesting of elements, and agent choices are represented by plateaus, or sibling elements. Unlike ANSI/CEA-2018 tasks, 'do' and 'say' semantics do not support the expression of partial ordering—for this one needs to drop down into ANSI/CEA-2018 representation.

In Figure 3.9 the first sidekick utterance starts a sequence[2]. It contains a pair of sibling 'say' elements, meaning that there are two alternative player utterances. Thus, the "1" and "2" XML comments in Figure 3.10 highlight the two player choices afforded by this decomposition. The remaining utterances and actions have no alternatives, and are thus sequential, but there are two special cases illustrated here. The first is the childless 'do' element, which is simply translated into a decomposition step. The second is the childless 'say' element that references another subtree using the 'ref' attribute and the 'id' of the referred-to subdialogue (an 'id' becomes the name of a subgoal containing the element's children, i.e., the root of the subtree). By including a reference of this nature, we effectively join the two trees at the point of reference (the node labeled LeadIn). Thus, in this instance both branches of the dialogue tree lead to the same end.

Elements may also declare an optional restriction on which actor is allowed to perform the

[2]There is an initial player utterance not present in Figure 3.9 that is supplied through automatic dialogue generation. The omitted player utterance actually starts the dialogue.

```
<say actor="sidekick"
   text="Clearly. You destroyed at least twenty in Cape Town alone.">
  <!-- 1 -->
  <say actor="player" text="So? It was easier to see them that way.">
    <say id="LeadIn" actor="sidekick"
      text="No, it wasn\'t. And it was more difficult to follow them, too!">
      <say actor="player" text="Whatever. We still have to get past this one.">
        <do task="Escape"/>
      </say>
    </say>
  </say>
  <!-- 2 -->
  <say actor="player" text="Whoah, whoah, whoah. That was an accident. ...">
    <say actor="sidekick"
        text="And some nice, innocent walls had to suffer for it.">
      <say ref="LeadIn"/>
    </say>
  </say>
</say>
```

Figure 3.9: Partial *Tizona* representation of Ice Wall task model

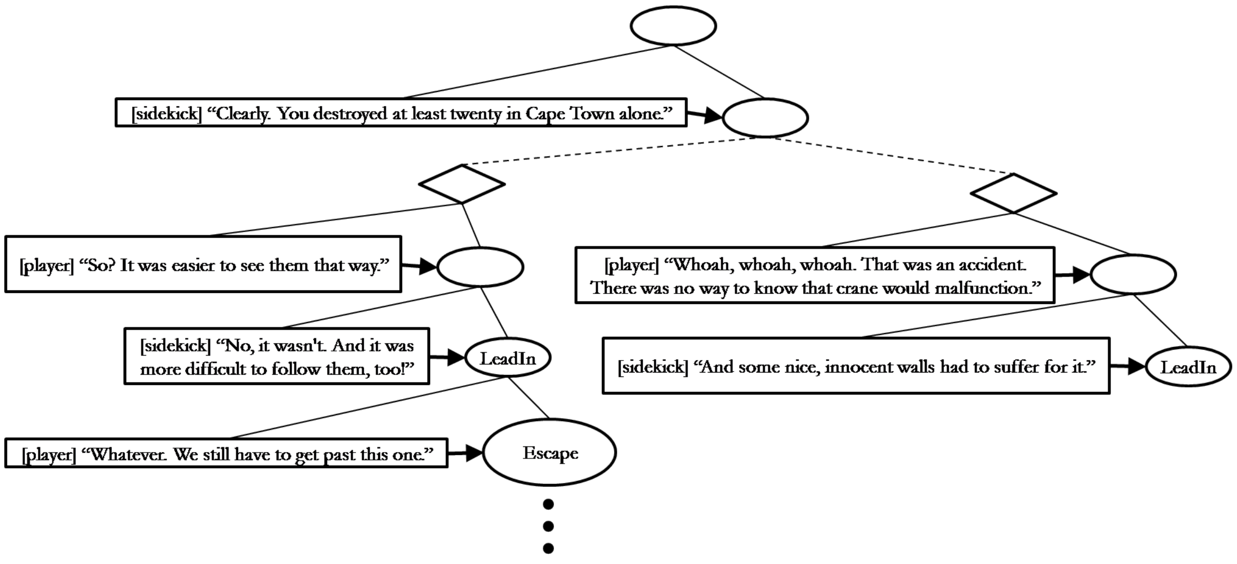

[sidekick] “Clearly. You destroyed at least twenty in Cape Town alone.”
[player] “So? It was easier to see them that way.”
[sidekick] “No, it wasn't. And it was more difficult to follow them, too!”
LeadIn
[player] “Whatever. We still have to get past this one.”
Escape
[player] “Whoah, whoah, whoah. That was an accident. There was no way to know that crane would malfunction.”
[sidekick] “And some nice, innocent walls had to suffer for it.”
LeadIn

Figure 3.10: Partial task model for Ice Wall level generated from Figure 3.9

action or produce the utterance. Standard ANSI/CEA-2018 elements may be included in the model after the *Tizona* elements (in a different namespace).

The listing in Figure 3.11 is a more complex example taken from Shelter level of *Secrets*, shown as a task diagram in Figure 3.7. Notice that, while this model incorporates more actions into the tree, it differs little in structure from the Ice Wall example in Figure 3.9. We nest 'do' and 'say' elements within each other, building up sequences and decompositions out of the given utterances and the referenced tasks.

The semantics of the 'do' element are to include the referenced task as a step and then include any child 'do' or 'say' tasks. It serves as a placeholder for a task that is defined elsewhere in the model. The task's identifier is stored as an attribute, and an appropriate reference is inserted upon translation. Each element can contain child elements of either type. Both elements may define an optional identifier that can be used to reference the element's subtree. During translation this has the effect of copying the subtree below the referenced element in place of the referencing 'do' or 'say' element.

The 'do' element requires an ANSI/CEA-2018 task with the given identifier to be present, and tasks can define decomposition steps that reference 'do' or 'say' trees by identifier, so all the features of ANSI/CEA-2018 are still available. For instance, both of the 'do' elements that reference BuildPillars have at least one 'say' element below them. The child 'say' elements will appear as utterances in a subtree of a step following the BuildPillars step, as shown in Figure 3.10. The full *Tizona* model of Shelter can be found in Appendix B.

3.3.2 Game API

While Tizona's core is independent from any specific game engine, we have produced a run time extension to Tizona for the Golden T game engine [6]. This extension, called Tizona-GT, maintains a synchronized state between the discourse engine and the game engine, and provides a streamlined programming interface for using Tizona in Golden T games. The extension manages the graphical display of player utterance menu and the NPC utterances. Several examples from *Secrets* exhibiting these graphical elements, such as Figure 2.2, can be found in Chap-

```
<say id="Shelter" actor="sidekick" text="Not so fast. I can\'t walk much further today,
and the weather\'s getting worse">
  <say actor="player" text="Okay. What should we do, then?">
    <say actor="sidekick" text="We need to build a shelter for the night">
      <say id="Floor" actor="player"
           text="Let\'s use pieces of that wreck to build a hut">
        <say actor="sidekick"
             text="Okay, the floor is flat already, so what should we build first?">
          <do task="BuildWalls">
            <say actor="sidekick" text="Do we want pillars at the front?">
              <applicable> !world.get("shack").hasPillar("right") </applicable>
              <say actor="player" text="Sure, let\'s go for it">
                <do id="Roof" task="BuildPillars">
                  <say actor="sidekick" text="Now all that\'s left is the roof!">
                    <do task="BuildRoof"/>
                  </say>
                </do>
              </say>
              <say actor="player" text="No, let\'s not have pillars">
                <say actor="sidekick" ref="Roof"/>
              </say>
            </say>
            <do task="_Roof_tree">
              <applicable> world.get("shack").hasPillar("right") </applicable>
              <say actor="sidekick" text="Glad that\'s over!"/>
            </do>
          </do>
          <do task="BuildPillars">
            <say actor="sidekick" text="They look cool, but we definitely need walls">
              <do task="BuildWalls">
                <say actor="sidekick" ref="Roof"/>
              </do>
            </say>
          </do>
        </say>
      </say>
      <say actor="player" text="We could build an igloo, I guess...">
        <say actor="sidekick"
             text="Ice pillars, maybe. But we don\'t have time for an igloo">
          <say actor="sidekick" ref="Floor"/>
        </say>
      </say>
    </say>
  </say>
</say>
```

Figure 3.11: Partial *Tizona* representation of Shelter task model from Figure 3.7

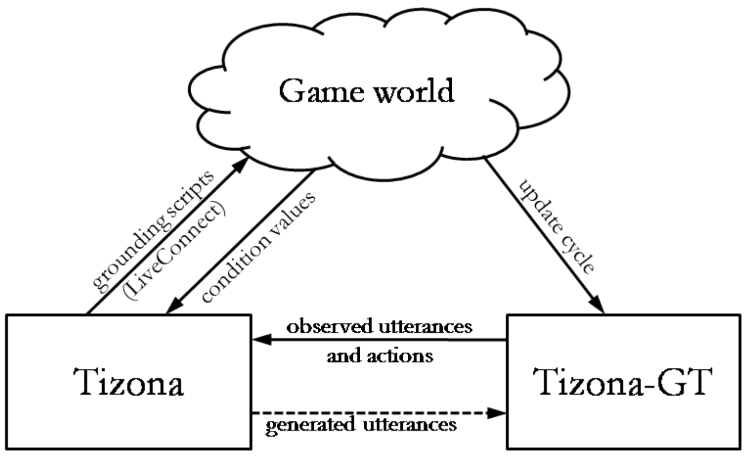


Figure 3.12: Communication between Tizona and game API

ter 2.1. Tizona-GT also handles the practical matter of cross-thread communication, as Tizona and Golden T run in different Java threads.

Tizona maintains a repository of data about the state of the game world in a thread-safe Java object. Tizona-GT exposes this repository to the game engine, allowing actions in the discourse engine to be reflected in the game engine's representation of the world. The extension also exposes a reference to the world state within the execution context of scripts from the task model. Figure 3.12 is a block diagram showing communication between Tizona and any extensions using the API and the world state.

ANSI/CEA-2018 tasks may optionally contain associated JavaScript code, called a "grounding script," that is connected to the game state using Java LiveConnect [14]. Figure 3.13 is a grounding script from the Ice Blocks level of *Secrets*. Grounding scripts are executed in order to accomplish the task. For instance, the grounding script in Figure 3.13 accomplishes the PushIceBlock task by moving the block into the water. We make use of Java's built-in JavaScript engine to embed the world state reference as a JavaScript variable called 'world.' Grounding scripts can use the 'world' variable to alter object states, thereby accomplishing the task in terms of the game state and providing implicit notification to the game engine. The game state can also be queried from preconditions and postconditions, allowing the system to react to changes in the game world that are not modeled explicitly as tasks—including changes not induced by conversation participants,

```
<task id="PushIceBlock">
  <input name="to" type="Packages.edu.wpi.secrets.objects.Area"/>
  <precondition>
    $this.to &amp;&amp; !$this.to.getAccessBlock().isInWater()
  </precondition>
  <postcondition sufficient="true">
    $this.to &amp;&amp; $this.to.getAccessBlock().isInWater()
  </postcondition>
  <script>
    if ( !$this.external )
      moveNPC("sidekick", $this.to.getAccessBlock().getLocation());
    $this.to.getAccessBlock().pushIntoWater();
  </script>
</task>
```

Figure 3.13: Example grounding script

like an ice block sliding into the water rather than being pushed.

When the player accomplishes a task, Tizona-GT creates a new instance of the task's type. The task's grounding script, if any, is then executed, and Tizona updates the discourse state. For example, suppose the player intends to push an ice block into the water in the Ice Blocks level of *Secrets*. Once the player has triggered the block in-game, a new task of type PushIceBlock (shown in Figure 3.13) will be created with appropriate variable values. This task will be sent to Tizona to have its grounding script executed (via LiveConnect), which updates the world state by changing properties of the ice block so it will appear in the water. After the grounding script has been executed Tizona will evaluate the postcondition, if any, and mark the task as 'succeeded' or 'failed' (or simply 'done' if there is no postcondition). Note that the game world state is updated through the grounding script, and the discourse state is updated by Tizona. Thus Tizona-GT establishes two-way state synchronization between the two engines.

Note that, in *Secrets*, avatar motion, i.e., moving left, right, up and down with the arrow keys, do not appear as actions in the task model or interaction history. Only actions performed through interaction with an object in the world are modeled. This is a level-of-abstraction modeling issue that is specific to each game. In *Secrets*, then, objects in the world can be used to perform actions "directly" by walking into them with the player avatar. Game objects that are activated by player movement (including constructs like triggers) subsequently create action instances that are observed by Disco so the action can be reflected in the discourse state.

3.3.3 Multi-way Conversations

Disco is limited by the ANSI/CEA-2018 standard to handle only interactions with exactly two participants, a player and a computer agent. While it is certainly possible to design games with only two characters—titles like *Myst* survive with the player as sole participant for long stretches—designers would probably find this design too limiting for mainstream games. Tizona extends the Disco model to support multiple participants in an interaction.

We account for extra participants by creating an instance of the discourse state for each *pair* of participants. We adapted the Disco interaction code to handle multi-way conversations that broadcast actions to all interactions that include the acting participant. A major downside of this approach is that it requires a separate task model to be authored for each pair of participants (though the models can certainly share tasks). Because the number of pairings grows with the square of the number of participants, the number of active participants in a single conversation should be limited in order to avoid creating a large number of task models. This can be accomplished by separating interactions by level, as in *Secrets*, or by restricting the number of participants that are active at a given time, e.g., exclusion by distance from the player.

Conversations can have participants added and removed at any time. A player can therefore engage in multiple *sequential* interactions with different agents in the same level, even leaving the conversation and returning to it. Dialogue state is stored in the conversation, so agents that leave and return will retain state from previous interactions, thus "remembering" where the conversation left off. This does lead to participant which have a "frozen memory," which—though better than no memory—is slightly unrealistic. Serial conversations of this nature could be improved by modeling agents leaving and returning within the task model so that characters can respond appropriately when the conversation resumes, e.g., asking why the player has been away for so long.

Agents can also participate *simultaneously* in a multi-way interaction, as shown in the Walrus Cave level of *Secrets*: the three actors (player, sidekick, and walrus) are aware of the others' actions and can respond to either of the other participants at any time if there is an appropriate utterance or action. The three task models necessary for this three-way interaction are included in Appendix B.

3.4 Additional Benefits

While our primary goal in using task models is to create a unified representation for talking and doing, this decision also brings some additional benefits to game authoring due to the connection with other work based on collaborative discourse theory.

3.4.1 Interruptions

When humans switch topics mid-conversation, they will often return to the original topic to complete the conversation. This behavior is accounted for by the *focus stack* in collaborative discourse theory, and is implemented in Disco. In Disco the participants' currently active shared goal is on top of the stack, but new goals can be pushed onto the top of it. When a goal is completed, it can be popped off the stack. The ability to handle interruptions – such as bringing up a topic or goal from an unrelated portion of the task model – and return to the original subject of the interaction once the interruption is complete comes as a natural side effect. The unrelated goal will be pushed onto the stack until it has been accomplished, and after popping it from the stack the focus will return to whatever goal previously held the focus of interaction.

Interruptions are characterized by "discontinuities" in the focus stack. Because each goal is pushed onto the stack in turn, there is a natural stratification: top-level goals are generally at the bottom of the stack, followed by their subgoals, the subgoals' subgoals, and so on. Whenever the relation between any two adjacent items in the stack is other than goal-subgoal, that discontinuity represents an interruption.

Figure 3.14 shows a simple task model and a representation of the focus stack at three different points in time. Initially the stack has the task A and its subgoal B in it. These tasks have a goal-subgoal relationship, so there is no interruption present. After some time has passed (stack 2), the task X has come into focus above B. However, B and X have no goal-subgoal relationship,

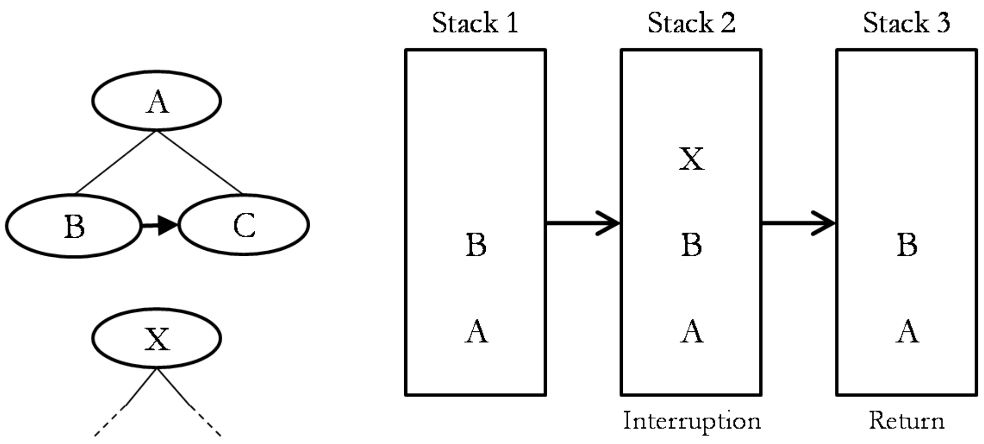


Figure 3.14: Diagram of a stack with interruption and return

so X is an interruption. Once X is accomplished (perhaps pushing and popping its subgoals, if any), it is popped off the stack (stack 3) and focus returns to B.

Most dialogue tree-based systems cannot support unplanned interruptions, because they do not have a focus stack. While there may be scripted interruptions, it is generally not possible to have, e.g., side conversations in a modal system. Interruptions simplify the handling of unrelated events, like random attacks. Interruptions also open the door to previously impractical level structures. A simple example of such a structure would be a detective game where the player and sidekick are always in conversation about the solution to the mystery, but this conversation can be interrupted other NPCs and affected by the information they provide.

Figure 3.15 shows an interaction history from the Ice Wall level of *Secrets* that includes an interruption (see Appendix B for the level's task model). At the moment shown in figure, the player has a choice to introduce a new topic of conversation by choosing a new top-level goal (choice [2]) from the utterance menu. Disco recognizes that this new goal, SillyConversation, is not related to the one below it on the stack by a goal-subgoal relationship, and therefore it must be an interruption. The second half of the figure shows the interaction history after the sub-dialogue is completed. We see that SillyConversation was popped off the stack and the actors resumed discussion of how to surmount the wall.

```
> history
[ ] -live
   Player says "A wall. Why does it always have to be walls? I hate walls!"
   Sidekick says "Clearly. You destroyed at least twenty in Cape Town alone."
   [Achieve PreCon] -live
      Player says "Whoah, whoah, whoah. That was an accident. There was no way to know
that crane would malfunction."
      [ ] -live
         Sidekick says "And some nice, innocent walls had to suffer for it."
         [ ] -live

> say
[1] Whatever. We still have to get past this one.
[2] Actually, there's something I've been meaning to tell you...
 >> 2

...

> history
[ ] -live
   Player says "A wall. Why does it always have to be walls? I hate walls!"
   Sidekick says "Clearly. You destroyed at least twenty in Cape Town alone."
   [Achieve PreCon] -live
      Player says "Whoah, whoah, whoah. That was an accident. There was no way to know
that crane would malfunction."
      [ ] -live
         Sidekick says "And some nice, innocent walls had to suffer for it."
         [Achieve SillyConversation] -done -interruption
            Player says "Actually, there's something I've been meaning to tell you..."
            Sidekick says "Really? Now is the time to do this?"
            [ ] -done
               Player says "Yes. I want you to know that..."
               [ ] -done
                  Player says "Umm... it's very cold out here."
                  [ ] -done
                     Sidekick says "How informative."
         [ ] -live
            [Achieve LeadIn] -live
               Player says "Whatever. We still have to get past this one."
               [ ] -live
                  [Get past it] -live
                     [Player says let's get past it somehow] -live <-focus
                        Sidekick says "How should we get past it?"
```

Figure 3.15: Interaction history before/after interruption and utterance menu with interruption choice

3.4.2 Automatic Dialogue Generation

We also find benefits in reducing the labor of dialogue creation through automatic dialogue generation. We will first explain the intuition of this approach and then give a more technical explanation. Automatic dialogue generation is made possible as a result of combining simple AI planning (implemented in Disco) with lightweight natural language semantics we draw from collaborative discourse theory [19]. We expand the dialogue tree to find the next live steps, and then the semantics determine a set of appropriate utterances (for either the player or the NPC).

Many different sets of utterances can be generated from a single task model. In this respect, we obtain compactness of representation in the same way that regular expressions simplify pattern matching by representing many possible matching strings at once. Use of automatic generation can thus reduce the labor involved in creating dialogues.

For example, suppose the current plan tree includes an instance of the action MoveIceBlock with unbound parameters for the actor (player or agent) and the ice block to be moved. This situation suggests two appropriate utterances, which supply the unbound variable values. An actor might reasonably ask at this point in the collaboration “Which ice block?” and “Who should move it?” These utterances *do not* need to be explicitly coded into the task model; they can be automatically generated from the semantics of the model.

```
> history
[Achieve CrossRiver] -live
   [Get to the first island by making a bridge] -done
      Player push an ice block into the water. -succeeded
      [Player walk to the first island]
         Sidekick says "Please walk to the first island."
   [Get to the second island] -live
      [Player says let's get to the second island] -live <-focus
         Sidekick says "How do you want to get to the second island?"
   [Get to the far side]

> say
[1] Let's get to the second island by making a bridge
[2] It's too far for me to swim. There's a rope, though.
[3] Let's not get to the second island
```

Figure 3.16: Example generated dialogue and interaction history

Another example of automatically generated dialogue is choosing a goal decomposition. When the task model includes several alternate methods of accomplishing a goal, an agent will automatically utter a “how” query. For example, once the goal named GetTo from Figure 3.2 has all its variables bound, an agent might ask “How to you want to achieve GetTo?” because there are three decompositions defined in the task model. The interaction history in Figure 3.16 shows the use of automatic dialogue generation in *Secrets of the Rime*: all three entries in the player menu have been automatically generated, corresponding to three alternative decomposition choices. This instance makes use of text replacement to improve the quality of utterances, as explained later in this section. The Ice Blocks level, which contains the example of Figure 3.16, relies exclusively on automatic dialogue generation for player and agent utterances.

Lightweight Utterance Semantics

Automatic dialogue generation is an outgrowth of using collaborative discourse theory as a foundation. Disco includes a library of task types with built-in semantics drawn from collaborative discourse theory [19]. Because the semantics of these utterance types is relative to a task model, they can be automatically chosen (generated) based on the current discourse state to algorithmically generate appropriate player and agent utterances. In addition to a general Say action type to represent utterances without semantics, Disco provides implementations for the following utterance types with the semantics (and an example) indicated:

- Propose.Should(?goal): initiate execution of a specified task (“Let’s achieve ?goal”)
- Propose.ShouldNot(?goal): stop or cancel execution of a task (“Let’s not ?goal”)
- Propose.What(?action, ?parameter, ?value): provide the value for an unbound variable (“The ?parameter of ?action is ?value”)
- Propose.Who(?who, ?action): propose which participant should execute a task (“I’m going to ?action”)
- Propose.Achieved(?goal): declare a task to have been successfully achieved (“?goal is achieved”)

- Propose.How(?goal, ?method): choose a decomposition (“Let’s achieve ?goal by ?method”)

There are also corresponding Ask utterance types, which have semantics of a question that is answered with the corresponding Propose utterance. One example is in climbing the wall in the Ice Wall level of *Secrets*: after the player chooses the decomposition using a Propose.How utterance (“Let’s get over it by climbing”), the sidekick will produce an Ask.Who utterance (“Who should give a boost over the wall?”) to determine which participant should perform the next action. To continue, the player can choose a Propose.Who utterance (“Give me a boost?” or “Up you go”) from the utterance menu to specify which of them should act. The player could also perform the action directly, which moots the question. Using these utterance types it is possible to model a large proportion of natural collaborative communication.

Dialogue Customization

While convenient, the text produced by automatically generated dialogue is almost certainly not good enough for production game use. Disco-generated dialogue is simple and includes task model identifiers. Nevertheless, a game can be prototyped quickly and incrementally using machine-generated utterance text until more polished, human-authored text is available. Machine-generated text can later be replaced by human-authored text or even voice-over audio. Disco provides two convenient methods for improving the default machine-generated text for builtin utterance types. All three levels of printing are illustrated in Figure 3.17, starting with the default level, which is used if neither of the other methods is employed by the author. Default strings are generally a description of the goal, slot values, and chosen decomposition, if any.

The first customization method uses native Java format strings, which are similar to formatted printing facilities in most other programming languages. Figure 3.18 lists format strings for a task and three decompositions in the Ice Blocks level of *Secrets*. Format strings for tasks can include a number of locations to insert text representing the values that fill a task’s slot values in order of definition. The resulting formatted text is much more natural and reads more or less like a normal English sentence, though it is perhaps a bit stilted. Not all constructions using format strings work out so well, particularly when the tense or number is different than the author expected.

Default:	`let's achieve GetTo on island1 and island2 and island1block by rope`
Formatted:	`let's get to the second island using the rope`
Translated:	`it's too far for me to swim. There's a rope, though...`

Figure 3.17: Three levels of printing for GetTo goal

```
GetTo@format = get to the %2$s
water@format = by swimming
bridge@format = by making a bridge
rope@format = using the rope
```

Figure 3.18: Properties for formatted printing

The second method of customizing machine-generated utterance text uses a translation file that maps specific machine-generated responses to human-authored text, a process that could easily be extended to audio files. The final line of Figure 3.17 shows the final utterance text. It is clearly part of a conversation and includes some mannerisms and hints about the character who is speaking. The same string can also be translated differently in different goal tree contexts to give more variety or specifics. The dialogue customization files used in the *Secrets of the Rime* can be found in Appendix B.

This approach is similar to that used in simple source-code internationalization: consider the source text to be another language, "computerese," to be translated into the native language of the player. Tizona is thus amenable to rapid prototyping of dialogue which will later be replaced with content authored to fit the final level. By judicious use of these dialogue generation and customization techniques, utterances can be included in development prototypes from the beginning of the design process and only customized later on, removing some potential bottlenecks in content creation.

3.4.3 Plan Recognition

In addition to planning for dialogue generation, another standard AI technique we get the benefit of by using task modeling is *plan recognition.* Plan recognition attempts to recognize patterns of actions and match them to decomposition steps in order to identify the goal an actor is working toward. This reduces the number of utterances needed for collaboration [11].

```
[Achieve CrossRiver] -live
   Player says "We need to get to the other side."
   [Get from the near side to the first island by making a bridge] -live
      [Player says let's get from the near side to the first island somehow] -done
         Sidekick says "How do you want to get from the near side to the first island?"
      Player says "Let's get from the near side to the first island by making a bridge."
      Sidekick push an ice block into the water. -succeeded
      [Player walk to the first island] -live <-focus
         Sidekick says "Please walk to the first island."
   [Get from the first island to the second island]
   [Get from the second island to the far side]
```

(a)

```
[Achieve CrossRiver] -live
   Player says "We need to get to the other side."
   [Get from the near side to the first island by making a bridge] -live <-focus
      [Player says let's get from the near side to the first island somehow] -done
         Sidekick says "How do you want to get from the near side to the first island?"
      Player push an ice block into the water. -succeeded
      [Player walk to the first island] -live <-focus
         Sidekick says "Please walk to the first island."
   [Get from the first island to the second island]
   [Get from the second island to the far side]
```

(b)

```
[Achieve CrossRiver] -live
   [Get from the near side to the first island by making a bridge] -live
      Player push an ice block into the water. -succeeded
      [Player walk to the first island] -live <-focus
         Sidekick says "Please walk to the first island."
   [Get from the first island to the second island]
   [Get from the second island to the far side]
```

(c)

Figure 3.19: Alternative interaction histories illustrating benefit of plan recognition

The interaction histories shown in Figure 3.19 illustrate the benefits of plan recognition in the Ice Blocks level of *Secrets of the Rime*. All three interactions achieve the same goal, but in (a) three utterances must be exchanged while (b) uses only two and (c) uses none. In (a), no plan recognition is performed: the "push ice block" action is accomplished as expected and is attached to the tree. Note that in this instance the player and sidekick engaged in a conversation to determine what action is expected. The player chose which decomposition to use via utterance, leaving "push ice block" as the only live action. The sidekick then performed this action when given the chance. Without plan recognition, this conversation would be mandatory in order for the sidekick follow the collaboration.

However, as we see in (b), Disco allows us to eliminate much of this dialogue, e.g., if the player wants to avoid it. Here the player first introduced the CrossRiver goal by selecting an option from the utterance menu and then implicitly chose the decomposition by performing the "push ice block" action. This action was recognized as the first step in the 'bridge' decomposition for GetTo, which was a live goal, so it was placed beneath the appropriate decomposition. That decomposition then became selected. The initial utterances between player and sidekick relate to intent, or focus in the goal tree, but the player chose not to explicitly answer the sidekick's question. Instead, the player performed a "push ice block" action.

In Figure 3.19(c), the player simply performed the "push ice block" action at the start. This action was recognized as a step in the plan for GetTo, under the CrossRiver goal. It is the first step of the decomposition for making a bridge, and it fits at no other place in the model, so it was placed beneath the appropriate decomposition. That decomposition then became selected, and the CrossRiver goal became 'live'. This instance leverages plan recognition to the fullest, eliminating all the utterances encountered in (a) and (b).

Plan recognition can be used to simplify level design, as authors can simply associate a task with a player action. This is sufficient to allow Disco to attach the action once performed and update the discourse state, and it allows the author to handle cases where the player ignores dialogue in a graceful manner.

3.4.4 Failure Modeling

The ANSI/CEA-2018 standard includes the notion of task failure. This intuitive concept simplifies the modeling of tasks with many failure modes and makes explicit the representation of "wrong turns" in exploration. When a task fails (when the task is done but its postcondition is not satisfied) it remains in the interaction history with a 'failed' flag, and Disco allows the rest of the interaction to continue. Disco also performs an 'automatic retry' when there are multiple decompositions, allowing the player or NPC to select a different decomposition to achieve the same goal. Without the notion of failure it would be necessary to create many alternative decompositions with hidden variables that restrict their applicability based on success, essentially recreating this system manually for each task model.

In the event that one of multiple decompositions fails, Disco will create a new instance of the goal and allow the player or NPC to select another decomposition (a “fail-over”) to continue trying to accomplish the current goal. This is reflected in the interaction history, as seen in Figure 3.20. Fail-over functionality also illustrates the utility of task models for modeling the *structure* of goals and actions rather than the specific *sequence* they can be accomplished in, though sequences can also be represented. Flexible level design will likely result in a large number of valid sequences that follow the same structure, and task modeling is well-suited to describing such flexible action patterns.

```
> history
...
   [ ] -live
      [Get past it] -live
         Player says "Let’s get past it."
         [Player says let’s get past it somehow] -live <-focus
            Sidekick says "How should we get past it?"

> say
[1] I say we climb over it.
[2] We could tunnel under it.
[3] No wall is an island. Can we go around?
[4] An ice wall? I’ve got the matches if you’ll find the kindling.
 >> 2

...

> history
...
   [ ] -live
      [Get past it by digging under] -failed
         Player says "Let’s get past it."
         [Player says let’s get past it somehow]
            Sidekick says "How should we get past it?"
         Player says "We could tunnel under it."
         Sidekick says "Are you crazy? There’s a reason it’s called permafrost: it’s
permanent."
      [Get past it] -live <-focus

> say
[1] I say we climb over it.
[2] No wall is an island. Can we go around?
[3] An ice wall? I’ve got the matches if you’ll find the kindling.
```

Figure 3.20: Partial interaction showing failure modeling in Ice Wall level from *Secrets*

Figure 3.20 shows a sequence of interaction histories from a portion of the Ice Walls level of *Secrets*. The first interaction history is the discourse state after the player has proposed the goal of getting past the wall, and the subsequent menu of player utterances has entries for all four of the goal's decompositions (see Appendix B for the full task model). The "get past it" goal (to circumvent a wall) has a postcondition that both actors should be standing on the far side of the wall, so a successful decomposition must end with both actors standing on that side.

The second interaction history is the discourse state after unsuccessfully trying one of these decompositions. All steps of the decomposition—in this case, a single utterance—were executed, but both actors remained on the near side of the wall. This state does not satisfy the postcondition, meaning that the goal was unsuccessful (not achieved). Therefore, the "digging under" decomposition was marked as "failed," while the overall goal of "get past it" remained active. Thus the second menu of player utterances includes the three untried decompositions, allowing the player to try an alternative tactic. Disco's failure-driven retry mechanism will continue until either goal is accomplished, all decompositions are exhausted, or the player stops pursuing the goal.

Chapter 4

Evaluation

There are two main challenges in evaluating a tool such as Tizona. The first challenge is a lack of established metrics for the effectiveness of tools: existing metrics for software products measure code quantity and performance rather than the quantity and performance of code based on the measured code [18, 21]. This leaves few options apart from user studies. The second challenge is the number of resources required to run a significant user study. We propose suitable metrics to overcome the first obstacle, and a hypothetical analysis to overcome the second one.

The ideal evaluation study for Tizona would be to assemble two equally skilled game development teams to work from the same design documents, where one team would use Tizona and the other would use modal technologies. If given the same art assets and game engine to begin with, the relative quality of output and speed of production of each team would provide some value measurement of the tool.

Due to the time and resources it would require, such a study is impractical for this project. In its place we present a comparison of development methods and careful analysis of the actual development process involved in producing the game *Secrets of the Rime* using Tizona. This game was produced in parallel with Tizona as a technology demo. Thus, we have a reference for modeling development with Tizona, though this reference comes from a team that is intimately familiar with Tizona. Development with modal technologies has many commercial references to draw from, as *Secrets* is similar in design to most adventure games.

For this analysis we assume use of the freeware Golden T Game Engine [6] and the existing Tizona extension for it that is described in Chapter 3. However, we will not assume that any development environments for task modeling are available. We also assume that the same art assets, such as map tiles and character sprites, are used for both projects and thus are not a factor. The following sections present a set of metrics we will use in comparing development technologies and an analysis of the two development methodologies considered, i.e., using Tizona and using modal technologies. We then examine the impact of Tizona on development and end product.

4.1 Metrics

We use standard software development metrics for comparison between the two development methods. We use two metrics, lines of code and number of classes, which relate to the size and complexity of the product [18]. Because the art assets, development language, and game design are the same in both cases, metrics such as function points [21] are mostly irrelevant. However, some of the game mechanics are not possible to implement using modal technologies, reducing the function point count of the modal version. The most notable mechanic of this sort is the ability to speak while performing an action, which requires the capabilities of two modes at once.

Finally, in addition to quality statements such as the above function point assertion, we will comment on the modifiability of each project in terms of lines of code as a measure of effort.

4.2 Development with Tizona

Our analysis of the development of *Secrets* using Tizona is based on work completed while developing the tool in parallel. Development in parallel exposed potential shortcomings and allowed us to test the usability of Tizona for development. This means that the game was produced by an expert in using the tool, potentially reducing the amount of time required, however much time was also spent debugging Tizona and working out issues with the extension for the game engine.

Proceeding by level, we defined the actions in each level's task model, as well as Java classes

Level	**Actions**	**Objects**
Ice Blocks	Swim, PushBlock, ThrowRope, CatchRope	IceBlock, Rope
Ice Wall	Boost, Clamber, OpenDoor, Dig, MakeFire	Door
Shelter	PickUp, Place	Shard, Panel
Walrus Cave	Boost, ExamineStatue, ThawDiamond, HoldFountain, OpenWindow	Statue, Diamond, Fountain, Window

Table 4.1: Actions and objects in *Secrets*

for each object type, based on the level description. These actions and objects are listed in Table 4.1. Because the game is divided into levels, it was convenient to create a separate task model for each level. We also created a Java class for each level, extending the GTLevel class in the Tizona extension for Golden T. Figure 4.1 shows a simplified Unified Modeling Language (UML) class diagram for *Secrets* when developed using Tizona. GTLevel handles coordination of the game state and actors with Tizona, and it also provides default behaviors for movement, performing actions, and emitting utterances. Thus the level classes only need to include exceptions to the default behavior and level-specific setup information.

In the level initialization methods, we created as many instances of an object as necessary, assigned the proper locations and object state, then added the objects to the global world state. For instance, the Ice Blocks level requires three IceBlock objects, one heavier than the others, and a single Rope. The Walrus Cave requires one of each of the objects listed in table 4.1.

The abstract class GameEngine is provided by Golden T, and that GTLevel and InteractableObject are provided by Tizona. From this diagram we see that development with Tizona results in the production of 14 Java classes and interfaces in addition to the 4 task models. The object classes in Figure 4.1 are small, as scripts to perform actions with these objects are actually contained in the task model. All task models for the game are included in Appendix B.

The combined line count for *Secrets* Java code and task model files is a little less than 1,400 lines of code. If the game were modified by adding a new level, it would require a new task model and level class, as well as object classes if the level includes objects. Changing a puzzle within a level requires only that the task model be changed, with potential tweaks to the level setup and additional object classes if new objects are specified.

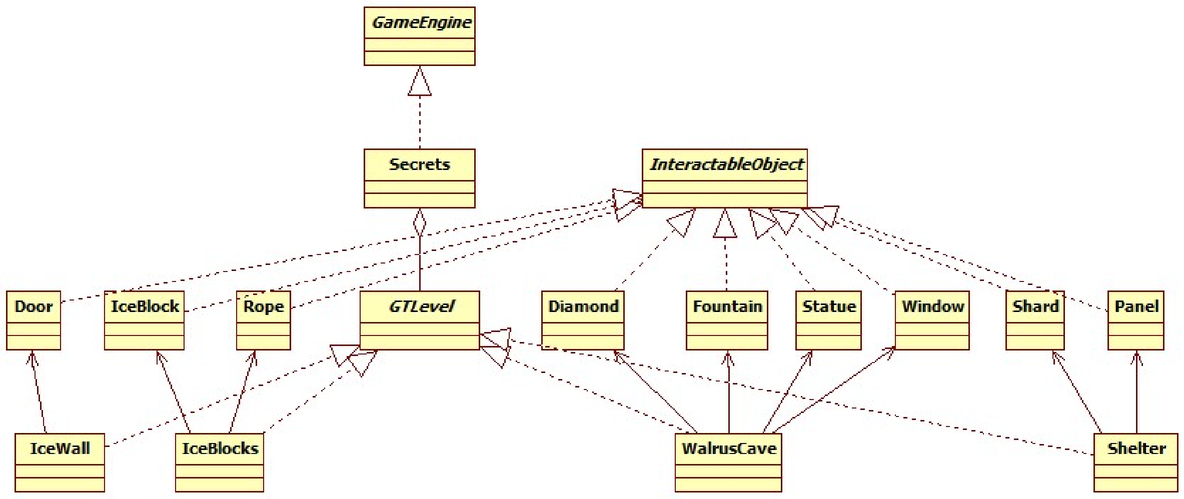


Figure 4.1: UML class diagram of *Secrets* when developed using Tizona

4.3 Modal System Development

For comparison, we now consider developing a modal system using the Golden T Game Engine directly, without Tizona.

If possible, we would have also chosen a preexisting dialogue engine to handle “talking mode,” but there are few readily available dialogue engines. The few available dialogue systems are embedded in a game engine, and most are not written in Java. There is no dialogue engine included in Golden T, nor is there a well-known third party solution for this engine. Essentially, this means that a development team that chooses not to license a game engine with dialogue support would have to implement it themselves. Furthermore, there is no standard representation for dialogue trees, so it would also be necessary either to define a file format or represent the trees in Java. We assume that a simple serialization class for dialogue trees and a dialogue engine would be produced to meet the needs of *Secrets*’ design.

Making good use of Golden T’s features, we would define a base class for levels to handle movement in “talking mode” and dialogue in “acting mode,” as well as to manage the transition between modes. Figure 4.2 shows a simplified UML class diagram for *Secrets* when developed without Tizona. The handling of each mode would take place in separate classes, called TalkMode and ActMode. The abstract classes GameEngine and GameObject, as well as the Sprite class, are provided by Golden T. However, player and NPC types, with specializations for each character, would be necessary to implement. From this diagram we see that a conservative estimate involves the creation of 24 types. Developers of a modal system would need to re-implement the default behaviors, as well as player action and utterance interfaces, that are specified in the Tizona extension for Golden T.

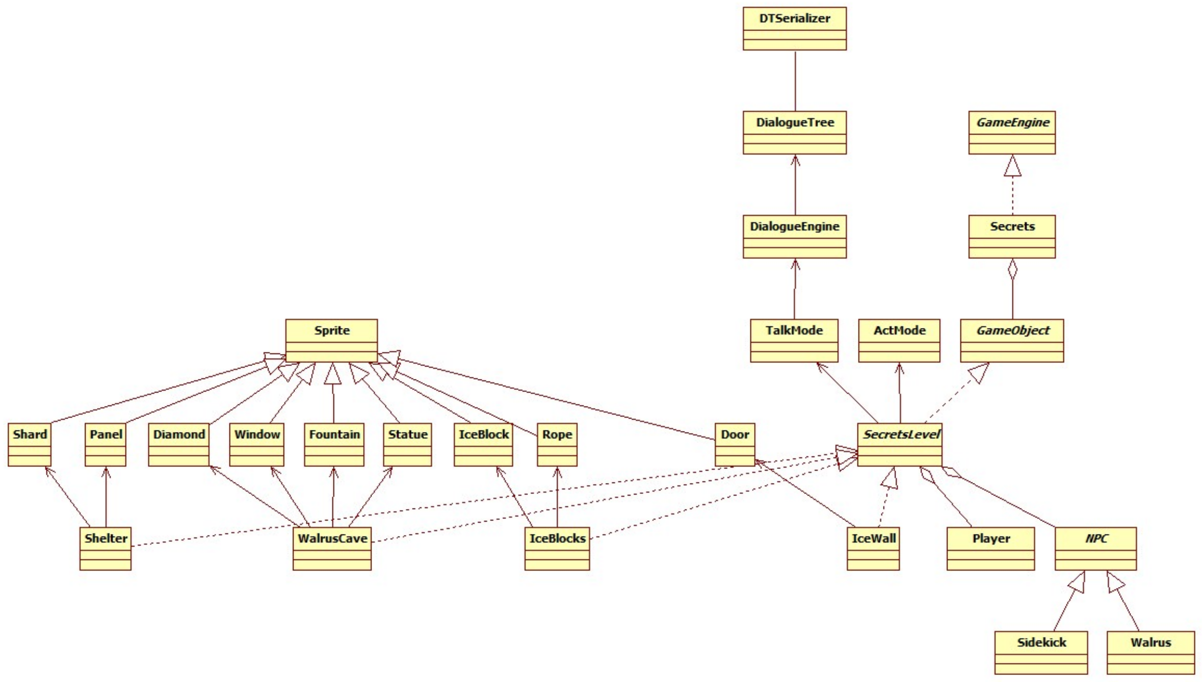


Figure 4.2: UML class diagram of *Secrets* when developed modally

Much of the implementation is straightforward, though the levels require occasional adaptation to fit the limitations of modal systems. For instance, it is likely that the Walrus Cave level would be adapted slightly to fit a fetch-quest design where player and sidekick can only converse with the walrus when standing directly in front of him. Commercial adventure games such as *Golden Sun* and *Beyond Good and Evil* follow this paradigm, where the game switches between talking mode and acting mode frequently to intermingle talking and doing.

To estimate lines of code for modal development, we begin by counting lines in the existing (Tizona-based) *Secrets* Java classes. We must also account for re-implementing behaviors, so we

	Existing Engine	New Engine
Tizona	1,400	1,400
Modal	1,900	3,700
Code reduction	26%	62%

Table 4.2: Comparison of development methods

include the Golden T extension to Tizona in this count, as well as the actor modeling classes. After including the extension, our estimate is approximately 1,900 lines of code. Finally, to reflect the implementation of a dialog engine, we include the core classes from Disco. These core classes form the basic Disco engine but not advanced features, so this is a conservative estimate of the effort required. This estimation gives a total of over 3,700 lines of code.

If we were to modify the game, adding a new level would require a new level class, as well as object classes if the level includes objects. Changing a puzzle within a level would require that program flow in the level classes—and potentially some object classes—be changed. Even if the level classes include comments detailing connections to the level's puzzle, it might be unclear whether the new code matches the design of the new puzzle.

4.4 Comparison

If we compare lines of code between projects, as shown in Table 4.2, it appears as though development with Tizona can reduce lines of code by 62%. However, this number includes development of a dialogue engine. After the initial implementation, a development team would most likely opt to update the engine for new projects rather than write a new engine. If we use the lower estimate for modal development—that is, excluding the dialogue engine—we find that development with Tizona can reduce lines of code by 26%.

In addition, games authored using Tizona use an abstract representation (a task model) to define the game world and possible actions. We assert that this abstraction allows developers to prototype games more quickly, much like working in a higher-level language. And as described above, changes to a level may only require changes to the task model, which has a direct and explicit connection to the level's puzzle. Thus we would expect to find an analogous decrease in development time for games developed with Tizona.

4.5 Play Testing

We performed an informal user study on *Secrets of the Rime* with two participants. One participant was an experienced computer game player from a college age group, and the other was a middle aged "casual gamer." The largest difference we observed was in approach to sidekick interaction. One participant was not interested in dialogue and chose the first utterance presented to them at any menu, while the other tended to forget that dialogue options were available, despite having used them moments before.

A common problem we observed was the choice of an utterance for which there was no agent reaction, such as "I'm going to open the door." Utterances of this nature are inert in that they do not change the world state. Users became confused when they chose one of these utterances and got no response from the NPCs. We subsequently fixed this problem by removing utterances with those semantics. However, participants had little difficulty in completing tasks that were initiated by a player utterance, followed by a sidekick action, and required further action from the player (such as being boosted over the wall in the Ice Blocks level and then being asked to open the door). This may indicate successful modeling of cooperative interactions.

Neither participant explored the dialogue options beyond what was needed to progress through the level, and they occasionally missed hints placed in NPC dialogue because they disregarded it. This can be attributed to user interface issues to some extent, though it is likely that the quality and style of authored dialogue also affects player willingness to carry on conversations. A third factor is highlighted by one participant—notably, the more experienced player—who thought that the sidekick "talked too much," or at least more than expected given other games they had played. At the time of play testing there was a bug in Tizona where agents would narrate their own actions, which may be what the participant was referring to; this bug has since been fixed. Because modern games typically use a modal system for dialogue, if at all, players are not used to constant interaction with NPC companions. This unfamiliarity is likely to be an obstacle in marketing a non-modal game or game engine.

Chapter 5

Related Work

To our knowledge, no work in games outside our own has addressed the integration of talking and doing technologies in a single representation. However, there are several related areas of research, such as interactive story generation, that extend the technologies we discuss or attempt to bridge the gap in other ways. It is useful to consider each related project as occupying a space on an authored-emergent spectrum, where at one end of the spectrum the behavior of the actors is totally controlled by the game author and at the other end it emerges from the actors (player, agents) themselves. A typical first-person shooter game with linear story-telling would fall at the extreme authorship end, for instance, while a flight simulator would fall at the extreme emergent end.

Like our work, Charles, Mead, and Cavazza [3] use hierarchical task networks for agent control and decision-making, though their approach is focused on task networks for individual agents instead of collaborative task networks. Their system displays strictly emergent agent behaviors and no explicit agent collaboration either with other agents or with players, placing it far to the emergent end of the spectrum. Their work is similar to our approach because agents have the capability to mix actions and utterances [2], although this is not through a general form and the system does not make use of semantics within the model, i.e., all dialogue must be modeled explicitly. Behavior in this system is non-deterministic: attempts to create a specific situation in the game by adjusting agent behavior are not sure to succeed. In the unified representation we propose, all agents behave deterministically[1], thus enabling authorship of specific situations or events.

Moving to the other end of the spectrum, there is the Wide Ruled project, a general framework based on the Universe story model [10]. Wide Ruled is focused entirely on goals set by the story author to compose a structured narrative that supports user interaction [20]. All aspects of the story are controlled through author goals and plot fragments, including character actions and environments. This approach is closer to our own in terms of goal focus (player and agent goals defined by the author), but the level of interactivity is low. There are no individual characters in Wide Ruled—all dialogue and actions are performed by the engine on a character's behalf. This approach could rightly be termed an "interactive storytelling" engine rather than a game or AI engine.

The Façade [12] project can be viewed as trying to combine the emergent agent behaviors of Charles, Mead, and Cavazza with the structured narrative of Wide Ruled in the form of author goals, or "story beats." While the characters of Grace and Trip are autonomous agents with their own goals most of the time, the Façade "drama manager" will at times reach in and insert goals or cause the agents to perform actions for dramatic effect. This structure addresses the problem of creating specific situations by overriding the emergent behavior at appropriate times and places Façade close to the center of the authored-emergent spectrum. Façade also bridges the talker/doer gap by adding a technology layer to coordinate talking and doing according to authored story beats. The approach is distinct from the unified model we propose in that shared plan elements are inserted by an external entity rather than pre-existing in the model.

One commercial example of talkers and doers is the game *Neverwinter Nights 2* (NWN2), which includes several "companions", which are sidekicks for your player character that can engage in both talking and fighting modes. However, the majority of agents encountered in the game are solely talkers or doers. Talking characters give out quests and information, and fighting (doing) characters attack on sight. The player must explicitly engage companions in dialogue, though they are sometimes included automatically in multiway conversations between the player

[1]This is because we impose a default order on partially ordered steps.

and other agents. At all other times, companions act as doers. This is what we called a *modal* integration approach, where an agent has the capacity to talk and to fight, but can only do one at a time.

The NWN2 game engine models attributes for the player character such as charisma, diplomacy, an “intimidation” factor, and position on a “good or evil” spectrum. These attributes have a bearing on whether some dialogue choices are available, and they may be changed by selecting some dialog choices, but outside of these effects, player actions have no effect on dialogue. For example, a player might steal all the items in an agent’s house, but unless that causes a shift in the player’s good-or-evil rating or the designer has included an extra script to detect the theft, the agent’s dialogue will not change.

Another illustration of a modal system is SIMDIALOG [15], a tool developed for producing educational games with extensive dialogue sequences. The SIMDIALOG tool is used to produce a set of dialogue trees whose execution is dependent on a user-defined set of hidden variables. Player dialogue choices can affect these hidden variables. Agent dialogue choices are selected based on the value of some variables and can also alter that value. Conversations end in a “terminal state”, which acts as a flag used by higher-level logic. A terminal state might indicate a new conversation to begin, some actions to be performed by an agent, or even the player’s death.

SIMDIALOG lies in the middle of the authored-emergent spectrum: while dialogue is authored by the developer, the player can have some measure of influence on the conversation by changing the hidden variables. Our approach also tracks state, but does so implicitly through the task model. One could recreate much of the SIMDIALOG system using Tizona by adding hidden variables to the task model.

The Forerunner project [13] at Worcester Polytechnic Institute implemented a “narrative engine” to direct the plot in a non-linear adventure game. Like Wide Ruled, Forerunner’s focus is on interactive story generation, but Forerunner incorporates more interactivity. The data structure used by the narrative engine in Forerunner is a form of and-or tree the authors call a story tree, and this structure bears a significant resemblance to hierarchical task networks [13, p. 14]. But while our work is intended to build on existing technologies and provide a platform for continued development, the Forerunner narrative engine is not generalized or easily extended and thus it is not likely to be useful as a tool.

Chapter 6

Conclusions and Future Work

Our contribution is twofold: firstly, we have defined a form and method for the unified representation of talking and doing agent behaviors in computer games. Secondly, we have developed a character-authoring tool called Tizona that enables game development with agents that are not limited by or divided into separate modes of interaction. Tizona, both as an authoring tool and as a run time environment, breaks down the dichotomy between talking technologies and doing technologies by handling both types of actions with the same run time engine. Tool and representation together serve to bridge the gap between talkers and doers, thereby making it possible to create richer and more interesting characters for computer games.

Our work also introduces several innovations in the authoring of agents, particularly through representation of agent behavior as data instead of code. Because the behavior description is stored in data files, it is more easily understood and altered—behaviors can be changed independently of changes to the game engine. Furthermore, the *Tizona* representation (which is based on an open standard) is analogous to a high-level declarative programming language, giving game developers greater expressive power.

What remains for future work is primarily to make the use of Tizona and its modeling language more practical. Specifically, an integrated development environment (IDE) for Tizona would increase ease of use. Note that an IDE for our representation could be implemented more easily than an IDE for modal technologies because task models are data files rather than code. A Tizona

IDE might include an editor for task model diagrams, allowing designers to "draw" the behavior of their agents. An IDE might also include a spreadsheet-like dialogue tree editor, which is a familiar environment for teams working with modal systems and thus might reduce the amount of adjustment necessary to use the new technology.

Non-modal interaction presents challenges in user interface design: while modal systems may halt to allow players to read every line of dialogue, non-modal systems must find an alternative method of presenting dialogue and actions to the player. Some method of determining acceptable timing and duration of utterances is also needed. Additionally, the increased complexity of selecting utterances in addition to normal play may be difficult to adapt to game consoles, which are limited to use of a controller. These issues will need to be addressed before any commercialization of Tizona or similar tools.

Using the *Tizona* format it would be possible to represent all required ANSI/CEA-2018 task models for a level using a single file, and to generate all of these models automatically using a script similar to that in Appendix D in order to alleviate the problem of authoring n^2 models for a conversation of n participants. The *Tizona* format could also be extended to support annotations and engine-specific metadata for tasks, thus reducing the amount of cross-referencing between files that human authors must do during development. Of course, these are all conveniences on the way to the ultimate goal of this project, which is to produce games with more characters that collaborate.

Appendix A

Interaction Histories for *Secrets*

The following interaction histories are examples of some of the many alternative interaction experiences possible for each level.

Ice Blocks

```
[Achieve CrossRiver] -succeeded
   Player says "We need to get to the other side."
   [Get to the first island by making a bridge] -done
      [Player says let's get to the first island somehow] -succeeded
         Sidekick says "How do you want to get to the first island?"
      Player push an ice block into the water. -succeeded
      [Player walk to the first island] -succeeded
         Sidekick says "Please walk to the first island."
   [Get to the second island using the rope] -succeeded
      [Player says let's get to the second island somehow] -succeeded
         Sidekick says "How do you want to get to the second island?"
      Player says "It's too far for me to swim. There's a rope, though..."
      [Sidekick swim to the second island] -accepted -done
         [Player says please swim to the second island] -done
            Sidekick says "Well, I can make it across. I'll send you a postcard
from the other side."
            Player says "Ok, in you go!"
         Sidekick swim to the second island.
      Sidekick throw the rope.
      [Player grab the rope] -done
         Sidekick says "Please grab the rope."
         Player grab the rope.
   [Get to the far side by swimming] -succeeded
      [Player says let's get to the far side somehow] -succeeded
         Sidekick says "How do you want to get to the far side?"
      Player says "Let's get to the far side by swimming."
      [Sidekick swim to the far side] -accepted -done
         [Player says please swim to the far side] -done
            Sidekick says "Should I go first?"
            Player says "Go ahead, race you there!"
      Player swim to the far side.
      [Sidekick swim to the far side] -accepted -done
         Sidekick swim to the far side.
```

Ice Wall

```
[Achieve IceWall] -done
   Player says "A wall. Why does it always have to be walls? I hate walls!"
   Sidekick says "Clearly. You destroyed at least twenty in Cape Town alone."
   [ ] -done
      Player says "Whoah, whoah, whoah. That was an accident. There was no way to know
that crane would malfunction."
      [ ] -done
         Sidekick says "And some nice, innocent walls had to suffer for it."
         [ ] -done
            [ ] -done
               Player says "Whatever. We still have to get past this one."
               [ ] -done
                  [Get past it by going around] -failed
                     [Player says let's get past it somehow] -done
                        Sidekick says "How should we get past it?"
                     Player says "No wall is an island. Can we go around?"
                     Sidekick says "It stretches on to the horizons. I heard that a
company called InfiniWall makes these."
                  [Get past it by climbing over] -succeeded
                     Player says "I say we climb over it."
                     [Sidekick give a boost over the wall] -accepted -done
                        [Player says "You should give a boost over the wall."] -succeeded
                           Sidekick says "Who should give a boost over the wall?"
                           Player says "Give me a boost?"
                        Sidekick give a boost over the wall.
                     [Player finish climbing over] -done
                        Sidekick says "Please finish climbing over."
                        Player finish climbing over.
                     [Player open the door] -done <-focus
                        Sidekick says "Please open the door."
                        [Achieve SillyConversation] -done -interruption
                           Player says "Actually, there's something I've been meaning to
tell you..."
                           Sidekick says "Really? Now is the time to do this?"
                           [ ] -done
                              Player says "Yes. I want you to know that..."
                              [ ] -done
                                 Player says "Umm... it's very cold out here."
                                 [ ] -done
                                    Sidekick says "How informative."
                        Sidekick says "Please open the door."
                        Player open the door.
                     [Sidekick execute Walk]
```

Shelter

```
[Achieve Shelter] -done
   Player says "All right, a clear path for once!"
   Sidekick says "Not so fast. I can’t walk much further today, and the weather’s
getting worse."
   [ ] -done
      Player says "Okay. What should we do, then?"
      [ ] -done
         Sidekick says "We need to build a shelter for the night."
         [ ] -done
            Player says "We could build an igloo, I guess..."
            [ ] -done
               Sidekick says "Ice pillars, maybe. But we don’t have time for an igloo."
               [ ] -done
                  [ ] -done
                     Sidekick says "Okay, the floor is flat already, so what should we
build first?"
                     [ ] -done
                        [Build the walls] -done
                           [Build the left wall] -done
                              [Sidekick find a panel to use] -done
                                 Player says "We need some walls."
                                 Sidekick find a panel to use.
                              Sidekick pick up a panel.
                              Sidekick place a panel on the left.
                           [Build the top wall] -done
                              Sidekick find a panel to use.
                              Sidekick pick up a panel.
                              Sidekick says "Ok."
                              [Sidekick place a panel on the top] -done
                                 Player says "Please place a panel on the top."
                                 Sidekick place a panel on the top.
                           [Build the right wall] -done
                              Sidekick find a panel to use.
                              Sidekick pick up a panel.
                              Sidekick place a panel on the right.
                        [ ] -done
                           Sidekick says "Do we want pillars at the front?"
                           [ ] -done
                              Player says "No, let’s not have pillars."
                              [ ] -done
                                 [ ] -done
```

```
                                    Sidekick says "Now all that's left is the roof!"
                                    [ ] -done
                                       [Make the roof] -done
                                          Sidekick find a panel to use.
                                          Sidekick pick up a panel.
                                          Sidekick place a panel on the roof.
```

Walrus Cave

Because there is a three-participant interaction in this level, thus three task models, we list here three interaction histories (one per model). A merged history printout would be useful future work for examining multi-way interactions.

Player and Walrus

```
[Achieve Convo] -done
  Player says "H-hello?"
  Walrus says "Ah, you have come at last. Welcome, great emissaries, to my hall!"
  [ ] -done
    Player says "Uh... hail, walrus! We seek the northern exit of your hall."
    [ ] -done
      Walrus says "What?! No one may exit the hall by the north gate unless they know
the secret of the Hall."
      [ ] -done
        Player says "Well, it's been great talking to you, great tusk, but we really
have to be going now."
        [ ] -done
          Walrus says "Not before you tell me the secret! I like the two of you, so
here's a clue:."
          [ ] -done
            Walrus says "Living threats your senses hone."
            [ ] -done
              Walrus says "The frozen twin smiling is seated."
              [ ] -done
                Walrus says "There upon a warming throne."
                [ ] -done
                  Walrus says "But you must not, or be defeated."
                  [ ] -done
                    Walrus says "Bring forth the adamantly polished."
                    [ ] -done
                      Walrus says "Hidden when walls were breached."
                      [ ] -done
                        Walrus says "Now... what is the secret?"
                        [ ] -done
                          Player says "A diamond?"
                          [ ] -done
```

```
                            Walrus says "Aha! The Adamant Request! Well done."
                            [ ] -done
                              Walrus leaves. -succeeded
```

Player and Sidekick

```
Sidekick says "Oh look, a talking walrus."
Player says "Uh... hail, walrus! We seek the northern exit of your hall."
Sidekick says "Seems to have a high opinion of himself for a talking pinniped."
Player says "Well, it's been great talking to you, great tusk, but we really have to
be going now."
Sidekick says "Finally. Thought he'd never shut up."
Player says "A whale?"
Player says "A hidden door?"
Player says "A diamond?"
Sidekick says "He creeps me out. Let's get out of here."
```

Sidekick and Walrus

```
[Achieve Convo] -done
   Walrus says "Ah, you have come at last. Welcome, great emissaries, to my hall!"
   [ ] -done
      Sidekick says "Oh look, a talking walrus."
[ ] -done
   Walrus says "What?! No one may exit the hall by the north gate unless they know the
secret of the Hall."
   [ ] -done
      Sidekick says "Seems to have a high opinion of himself for a talking pinniped."
Walrus says "Not before you tell me the secret! I like the two of you, so here's a
clue:."
Walrus says "Living threats your senses hone."
Walrus says "The frozen twin smiling is seated."
Walrus says "There upon a warming throne."
Walrus says "But you must not, or be defeated."
Walrus says "Bring forth the adamantly polished."
Walrus says "Hidden when walls were breached."
[Achieve Secret] -done
   Walrus says "Now... what is the secret?"
   [ ] -done
      Sidekick says "Finally. Thought he'd never shut up."
[Achieve AdamantRequest] -done
   Walrus says "Aha! The Adamant Request! Well done."
   [ ] -done
      Sidekick says "He creeps me out. Let's get out of here."
```

Appendix B

Task Models for *Secrets*

The following sections contain listings of the task models and associated files (for dialogue customization) used to create the game *Secrets of the Rime*, described in Section 2. The Ice Blocks model is in pure ANSI/CEA-2018 format. The remaining three task models are in *Tizona* format, which can be transformed into an ANSI/CEA-2018 format task model for use with Disco and Tizona using the script contained in Appendix D.

Ice Blocks

Task Model

```
<taskModel about="urn:cetask.wpi.edu:models:secrets:IceBlocks"
           xmlns="http://www.cs.wpi.edu/~rich/cetask/cea-2018-ext">

   <!-- Overall goal: get to the other side -->
   <task id="CrossRiver">
      <postcondition sufficient="true">
       on("player", farSide) &amp;&amp; on("sidekick", farSide)
      </postcondition>
      <subtasks id="cross">
         <step name="goIsland1" task="GetTo"/>
         <step name="goIsland2" task="GetTo"/>
         <step name="goFarSide" task="GetTo"/>
         <binding slot="$goIsland1.from" value="world.get('nearSide')"/>
         <binding slot="$goIsland1.to" value="world.get('island1')"/>
         <binding slot="$goIsland2.from" value="world.get('island1')"/>
         <binding slot="$goIsland2.to" value="world.get('island2')"/>
         <binding slot="$goFarSide.from" value="world.get('island2')"/>
         <binding slot="$goFarSide.to" value="world.get('farSide')"/>
      </subtasks>
   </task>

   <!-- Subgoal: get to a specific area -->
   <task id="GetTo">
      <input name="from" type="Packages.edu.wpi.secrets.objects.Area"/>
      <input name="to" type="Packages.edu.wpi.secrets.objects.Area"/>
      <postcondition sufficient="true">
       on("player", $this.to) &amp;&amp; on("sidekick", $this.to)
      </postcondition>

      <subtasks id="water" ordered="false">
         <step name="sidekickSwim" task="Swim"/>
         <step name="playerSwim" task="Swim"/>
         <applicable> $this.to != world.get("island2") </applicable>
         <binding slot="$sidekickSwim.from" value="$this.from"/>
         <binding slot="$sidekickSwim.to" value="$this.to"/>
         <binding slot="$sidekickSwim.external" value="false"/>
         <binding slot="$playerSwim.from" value="$this.from"/>
         <binding slot="$playerSwim.to" value="$this.to"/>
         <binding slot="$playerSwim.external" value="true"/>
      </subtasks>

      <subtasks id="bridge">
         <step name="pushBlock" task="PushIceBlock"/>
         <step name="walk" task="Walk"/>
         <applicable> $this.to != world.get("island2") </applicable>
         <binding slot="$pushBlock.to" value="$this.to"/>
         <binding slot="$walk.to" value="$this.to"/>
      </subtasks>

      <subtasks id="rope">
         <step name="sidekickSwim" task="Swim"/>
         <step name="throw" task="ThrowRope"/>
         <step name="catch" task="CatchRope"/>
         <applicable> $this.to == world.get("island2") </applicable>
         <binding slot="$sidekickSwim.from" value="$this.from"/>
         <binding slot="$sidekickSwim.to" value="$this.to"/>
         <binding slot="$sidekickSwim.external" value="false"/>
         <binding slot="$throw.external" value="false"/>
         <binding slot="$catch.external" value="true"/>
      </subtasks>
   </task>

   <!-- Primitive actions -->
   <task id="Walk">
      <input name="to" type="Packages.edu.wpi.secrets.objects.Area"/>
      <binding slot="$this.external" value="true"/>
      <postcondition sufficient="true"> on("player", $this.to) </postcondition>
   </task>
```

```
<task id="Swim">
   <input name="from" type="Packages.edu.wpi.secrets.objects.Area"/>
   <input name="to" type="Packages.edu.wpi.secrets.objects.Area"/>
   <script>
    if ( $this.external ) movePlayer($this.to.getWalkToLocation());
    else moveNPC("sidekick", $this.to.getWalkToLocation());
   </script>
</task>

<task id="PushIceBlock">
   <input name="to" type="Packages.edu.wpi.secrets.objects.Area"/>
   <precondition>
     $this.to &amp;&amp; !$this.to.getAccessBlock().isInWater()
   </precondition>
   <postcondition sufficient="true">
     $this.to &amp;&amp; $this.to.getAccessBlock().isInWater()
   </postcondition>
   <script>
    if ( !$this.external )
       moveNPC("sidekick", $this.to.getAccessBlock().getLocation());
     $this.to.getAccessBlock().pushIntoWater();
   </script>
</task>

<task id="ThrowRope">
   <script>
    if ( world.get("rope") != null ) {
       world.get("rope").setThrown(true);
       world.get("ropetrigger").activate();
     }
 </script>
</task>

<task id="CatchRope">
   <script>
    if ( world.get("rope") != null ) {
       world.get("rope").setThrown(false);
       world.get("player").getLocation().setLocation(world.get("rope").getLocation());
     }
 </script>
</task>

<!-- Useful functions for writing conditions -->
<script init="true">
   function player () { return getX("player"); }
   function sidekick () { return getX("sidekick"); }
   function getX (actor) { return world.get(actor).getLocation().x; }
   function on (actor, area) {
     return (area) &amp;&amp; (getX(actor) &gt;= area.getWalkToLocation().x);
   }
   var farSide = world.get("farSide");
</script>
```

```
</taskModel>
```

Properties File

```
@how = how do you want to

GetTo@format = get to the %2$s

water@format = by swimming
bridge@format = by making a bridge
rope@format = using the rope

rope@authorized = false

Swim@format = swim to the %2$s
Walk@format = walk to the %s
PushIceBlock@format = push an ice block into the water
ThrowRope@format = throw the rope
CatchRope@format = grab the rope

Swim@authorized = false

Swim@ProposeShouldNot = false
CrossRiver@ProposeShouldNot = false
```

Translation File

```
let's_achieve_CrossRiver = we need to get to the other side
let's_get_to_the_second_island_using_the_rope = it's too far for me to swim. There's
a rope, though...
should_I_swim_to_the_second_island? = well, I can make it across. I'll send you a
postcard from the other side
please_swim_to_the_second_island = ok, in you go!
should_I_swim_to_the_far_side? = should I go first?
please_swim_to_the_far_side = go ahead, race you there!
```

Ice Wall

Task Model

```
<model about="urn:cetask.wpi.edu:models:secrets:IceWall"
       xmlns="http://www.cs.wpi.edu/~rich/tizona"
       xmlns:d="http://www.cs.wpi.edu/~rich/cetask/cea-2018-ext">

  <!-- Introductory dialogue tree, including banter -->
  <say id="IceWall" actor="sidekick"
       text="Clearly. You destroyed at least twenty in Cape Town alone.">
    <say actor="player" text="So? It was easier to see them that way.">
      <say id="LeadIn" actor="sidekick"
           text="No, it wasn\'t. And it was more difficult to follow them, too!">
        <say actor="player" text="Whatever. We still have to get past this one.">
          <do task="Escape"/>
        </say>
      </say>
    </say>
    <say actor="player" text="Whoah, whoah, whoah. That was an accident. There was no
way to know that crane would malfunction.">
      <say actor="sidekick" text="And some nice, innocent walls had to suffer for it.">
        <say ref="LeadIn"/>
      </say>
    </say>
  </say>

  <d:task id="Escape">
    <d:postcondition sufficient="true">
      world.get("door").isOpen() &amp;&amp; (player() &gt;= walkToLocation.x)
    </d:postcondition>

    <!-- Climb over the wall -->
    <d:subtasks id="over">
      <step name="boost" task="Boost"/>
      <step name="clamber" task="Clamber"/>
      <step name="open" task="OpenDoor"/>
      <step name="walk" task="Walk"/>
      <binding slot="$clamber.external" value="!$boost.external"/>
      <binding slot="$open.external" value="!$boost.external"/>
      <binding slot="$walk.external" value="$boost.external"/>
    </d:subtasks>

    <!-- Dig under the wall -->
    <d:subtasks id="under">
      <d:step name="say" task="disco:edu.wpi.disco.lang.SayAgent"/>
      <d:binding slot="$say.text" value="'Are you crazy? There\'s a reason it\'s called
permafrost: it\'s permanent.'"/>
    </d:subtasks>

    <!-- Walk around the wall -->
    <d:subtasks id="around">
      <d:step name="say" task="disco:edu.wpi.disco.lang.SayAgent"/>
      <d:binding slot="$say.text" value="'It stretches on to the horizons. I heard that
a company called InfiniWall makes these.'"/>
    </d:subtasks>

    <!-- Cut/melt/explode through the wall -->
    <d:subtasks id="through">
      <d:step name="say" task="disco:edu.wpi.disco.lang.SayAgent"/>
      <d:binding slot="$say.text" value="'Are you kidding? There\'s no kindling here!'"/>
    </d:subtasks>
  </d:task>

  <d:task id="Boost">
    <d:script>
      actor($this.external).setMovable(false);
      other($this.external).setMovable(false);
      world.get("climbableWall").climb();
      actor($this.external).getLocation().setLocation(boosterLocation);
      other($this.external).getLocation().setLocation(boostedLocation);
    </d:script>
  </d:task>

  <d:task id="Clamber">
    <d:script>
      world.get("clamberTrigger").clamber();
      actor($this.external).getLocation().setLocation(clamberLocation);
      actor($this.external).setMovable(true);
      other($this.external).setMovable(true);
    </d:script>
  </d:task>

  <d:task id="OpenDoor">
    <d:script>
      if ( !$this.external ) moveNPC("sidekick", walkToLocation);
      if ( world.get("door") ) world.get("door").open();
    </d:script>
  </d:task>

  <d:task id="Walk">
    <d:postcondition>
      (actor($this.external) != undefined) &amp;&amp;
      (getX(actor($this.external)) &gt;= walkToLocation.x)
    </d:postcondition>
    <d:script>
      if ( $this.external ) movePlayer(walkToLocation);
      else moveNPC("sidekick", walkToLocation);
    </d:script>
  </d:task>

  <!-- Possible interruption -->
  <d:task id="SillyConversation">
    <d:precondition>
      (player() &gt;= walkToLocation.x) &amp;&amp; (sidekick() &lt; walkToLocation.x)
    </d:precondition>
    <d:subtasks id="silly_subtasks">
      <d:step name="say" task="disco:edu.wpi.disco.lang.SayAgent"/>
      <d:step name="rest_of_conversation" task="_SillyConversationContinued"/>
      <d:binding slot="$say.text" value="'Really? Now is the time to do this?'"/>
    </d:subtasks>
  </d:task>
```

```
  <say id="_SillyConversationContinued" actor="player"
       text="Yes. I want you to know that...">
    <say actor="player" text="I\'m the one who stole the diamonds.">
      <say actor="sidekick" text="Huh?">
        <say actor="player" text="Uh... never mind."/>
      </say>
    </say>
    <say actor="player" text="Umm... it\'s very cold out here.">
      <say actor="sidekick" text="How informative." />
    </say>
  </say>

  <d:script init="true">
    function player () { return getX("player"); }
    function sidekick () { return getX("sidekick"); }
    function getX (actor) { return world.get(actor).getLocation().x; }

    function actor (external) {
      if (external == undefined) return undefined;
      if (external) return world.get("player");
      return world.get("sidekick");
    }
    function other (external) {
      if (external == undefined) return undefined;
      if (external) return world.get("sidekick");
      return world.get("player");
    }

function rel(object, offX, offY) {
    location = object.getLocation();
    return new Packages.java.awt.Point(location.x + offX, location.y + offY);
  }

boosterLocation = rel(world.get("climbableWall"), -1, 0);
boostedLocation = rel(world.get("clamberTrigger"), -1, 0);
clamberLocation = rel(world.get("clamberTrigger"), 1, 0);
    walkToLocation = world.get("door").getLocation();
  </d:script>
</model>
```

Properties File

```
@someone = one of us
@how = how should we

Escape@format = get past it
over@format = by climbing over
under@format = by digging under
around@format = by going around
through@format = by sheer force
```

```
Boost@format =  give a boost over the wall
Boost@authorized = false

Clamber@format = finish climbing over
OpenDoor@format = open the door

IceWall@ProposeShouldNot = false
Escape@ProposeShouldNot = false
```

Translation File

```
let’s_achieve_IceWall = a wall. Why does it always have to be walls? I hate walls!

let’s_achieve_SillyConversation = actually, there’s something I’ve been meaning to tell you...
let’s_not_achieve_SillyConversation = Forget it

let’s_get_past_it_by_climbing_over = I say we climb over it
let’s_get_past_it_by_digging_under = we could tunnel under it
let’s_get_past_it_by_going_around = no wall is an island. Can we go around?
let’s_get_past_it_by_sheer_force = an ice wall? I’ve got the matches if you’ll find the kindling

I_should_give_a_boost_over_the_wall = up you go
you_should_give_a_boost_over_the_wall = give me a boost?
let’s_not_give_a_boost_over_the_wall = actually, I just remembered that I’m scared of heights
I’m_not_going_to_give_a_boost_over_the_wall = I’m not going to give you a boost
I’m_going_to_give_a_boost_over_the_wall = I’m going to give you a boost

please_finish_climbing_over = well, are you going to climb over or not?
```

Shelter

Task Model

```
<?xml version="1.0" encoding="utf-8"?>
<model about="urn:cetask.wpi.edu:models:secrets:Shelter"
       xmlns="http://www.cs.wpi.edu/~rich/tizona"
       xmlns:d="http://www.cs.wpi.edu/~rich/cetask/cea-2018-ext">

  <say id="Shelter" actor="sidekick" text="Not so fast. I can\’t walk much further
today, and the weather\’s getting worse">
    <say actor="player" text="Okay. What should we do, then?">
      <say actor="sidekick" text="We need to build a shelter for the night">
        <say id="Floor" actor="player"
             text="Let\’s use pieces of that wreck to build a hut">
          <say actor="sidekick"
               text="Okay, the floor is flat already, so what should we build first?">
            <do task="BuildWalls">
              <say actor="sidekick" text="Do we want pillars at the front?">
                <applicable> !world.get("shack").hasPillar("right") </applicable>
                <say actor="player" text="Sure, let\'s go for it">
                  <do id="Roof" task="BuildPillars">
                    <say actor="sidekick" text="Now all that\'s left is the roof!">
                      <do task="BuildRoof"/>
                    </say>
                  </do>
                </say>
                <say actor="player" text="No, let\'s not have pillars">
                  <say actor="sidekick" ref="Roof"/>
                </say>
              </say>
              <do task="_Roof_tree">
                <applicable> world.get("shack").hasPillar("right") </applicable>
                <say actor="sidekick" text="Glad that\'s over!"/>
              </do>
            </do>
            <do task="BuildPillars">
              <say actor="sidekick" text="They look cool, but we definitely need walls">
                <do task="BuildWalls">
                  <say actor="sidekick" ref="Roof"/>
                </do>
              </say>
            </do>
          </say>
        </say>
        <say actor="player" text="We could build an igloo, I guess...">
          <say actor="sidekick"
               text="Ice pillars, maybe. But we don\'t have time for an igloo">
            <say actor="sidekick" ref="Floor"/>
          </say>
        </say>
      </say>
    </say>
  </say>

  <d:task id="BuildWalls">
    <d:subtasks id="_walls">
      <d:step name="lwall" task="BuildWall"/>
      <d:step name="twall" task="BuildWall"/>
      <d:step name="rwall" task="BuildWall"/>
      <d:binding slot="$lwall.placement" value="'left'"/>
      <d:binding slot="$twall.placement" value="'top'"/>
      <d:binding slot="$twall.external" value="!$lwall.external"/>
      <d:binding slot="$rwall.placement" value="'right'"/>
    </d:subtasks>
  </d:task>

  <d:task id="BuildWall">
    <d:input name="placement" type="string"/>
    <postcondition sufficient="true">
      world.get("shack").hasWall($this.placement)
    </postcondition>
    <d:subtasks id="_wall">
      <d:step name="choose" task="ChoosePanel"/>
      <d:step name="pickUp" task="PickUpPanel"/>
      <d:step name="place" task="PlacePanel"/>
      <d:binding slot="$choose.external" value="$this.external"/>
      <d:binding slot="$pickUp.panel" value="$choose.panel"/>
      <d:binding slot="$pickUp.external" value="$this.external"/>
      <d:binding slot="$place.placement" value="$this.placement"/>
      <d:binding slot="$place.panel" value="$pickUp.panel"/>
      <d:binding slot="$place.external" value="$this.external"/>
    </d:subtasks>
  </d:task>

  <d:task id="ChoosePanel">
    <d:output name="panel" type="Packages.edu.wpi.secrets.objects.Panel"/>
    <d:script>
      $this.panel = getNearest("panel", actor($this.external));
      $this.success = ($this.panel != null);
    </d:script>
  </d:task>

  <d:task id="PickUpPanel">
    <d:input name="panel" type="Packages.edu.wpi.secrets.objects.Panel"/>
    <d:script>
      if ( !$this.external ) moveNPC("sidekick", $this.panel.getLocation());
      actor($this.external).pickUp($this.panel);
      $this.panel.pickUp();
    </d:script>
  </d:task>

  <d:task id="PlacePanel">
    <d:input name="placement" type="string"/>
    <d:input name="panel" type="Packages.edu.wpi.secrets.objects.Panel"/>
    <d:script>
      shack = world.get("shack");
      if ( !$this.external ) moveNPC("sidekick", rel(shack, 0, -1))
      if ( $this.placement == 'left' ) actor($this.external).setMovable(false);
      actor($this.external).dropCarriedObject();
      shack.place($this.placement, $this.panel);
      if ( $this.placement == 'top' ) other($this.external).setMovable(true);
    </d:script>
  </d:task>

  <d:task id="BuildPillars">
    <d:subtasks id="_pillars" ordered="false">
      <d:step name="pillar1" task="BuildPillar"/>
      <d:step name="pillar2" task="BuildPillar"/>
      <d:binding slot="$pillar1.placement" value="'left'"/>
      <d:binding slot="$pillar2.placement" value="'right'"/>
    </d:subtasks>
  </d:task>

  <d:task id="BuildPillar">
    <d:input name="placement" type="string"/>
    <postcondition sufficient="true">
      world.get("shack").hasPillar($this.placement)
    </postcondition>
    <d:subtasks id="_pillar">
      <d:step name="choose" task="ChooseShard"/>
      <d:step name="pickUp" task="PickUpShard"/>
      <d:step name="putDown" task="PlaceShard"/>
      <d:binding slot="$pickUp.shard" value="$choose.shard"/>
      <d:binding slot="$pickUp.external" value="$choose.external"/>
      <d:binding slot="$putDown.placement" value="$this.placement"/>
      <d:binding slot="$putDown.shard" value="$choose.shard"/>
      <d:binding slot="$putDown.external" value="$choose.external"/>
    </d:subtasks>
  </d:task>

  <d:task id="ChooseShard">
    <d:output name="shard" type="Packages.edu.wpi.secrets.objects.IceShard"/>
    <d:script>
      $this.shard = getNearest("shard", actor($this.external));
      $this.success = ($this.shard != null);
    </d:script>
  </d:task>

  <d:task id="PickUpShard">
    <d:input name="shard" type="Packages.edu.wpi.secrets.objects.IceShard"/>
    <d:precondition>
      ($this.external == undefined) || (!actor($this.external).isCarrying())
    </d:precondition>
    <d:script>
      if ( !$this.external ) { moveNPC("sidekick", $this.shard.getLocation()); }
      actor($this.external).pickUp($this.shard);
      $this.shard.pickUp();
    </d:script>
  </d:task>

  <d:task id="PlaceShard">
    <d:input name="placement" type="string"/>
    <d:input name="shard" type="Packages.edu.wpi.secrets.objects.IceShard"/>
    <d:precondition>
      ($this.external != undefined) &amp;&amp;
      (actor($this.external).getCarriedObject() == $this.shard)
    </d:precondition>
    <d:script>
      shack = world.get("shack");
      if ( !$this.external ) moveNPC("sidekick", rel(shack, 0, 1))
      actor($this.external).dropCarriedObject();
      shack.place($this.placement, $this.shard);
    </d:script>
  </d:task>

  <d:task id="BuildRoof">
    <postcondition sufficient="true">
      world.get("shack").hasRoof()
    </postcondition>
    <d:subtasks id="_roof">
      <d:step name="choose" task="ChoosePanel"/>
      <d:step name="pickUp" task="PickUpPanel"/>
      <d:step name="place" task="PlacePanel"/>
      <d:binding slot="$pickUp.panel" value="$choose.panel"/>
      <d:binding slot="$pickUp.external" value="$choose.external"/>
      <d:binding slot="$place.placement" value="'roof'"/>
      <d:binding slot="$place.panel" value="$pickUp.panel"/>
      <d:binding slot="$place.external" value="$choose.external"/>
    </d:subtasks>
  </d:task>

  <d:script init="true">
    function actor (external) {
      if ( external == undefined ) return undefined;
      if ( external ) return world.get("player");
      return world.get("sidekick");
    }
    function other (external) {
      if ( external == undefined ) return undefined;
      if  (external ) return world.get("sidekick");
      return world.get("player");
    }
    function rel(object, offX, offY) {
      location = object.getLocation();
      return new Packages.java.awt.Point(location.x + offX, location.y + offY);
    }
    function getNearest(prefix, actor) {
      var i = 1;
      var min = 1000;
      var nearest = null;
      var current = null;
      while ( (current = world.get(prefix + "" + i)) != null ) {
        if ( !current.isPickedUp() ) {
          dist = current.getLocation().distance(actor.getLocation());
          if ( dist &lt; min ) {
            nearest = current;
            min = dist;
          }
        }
       i++;
      }
      return nearest;
    }
  </d:script>
```

```
</model>
```

Properties File

```
@someone = we

Shelter@ProposeShouldNot = false

BuildPillars@format = put up some pillars
BuildWalls@format = build the walls
BuildRoof@format = make the roof

BuildWalls@ProposeShouldNot = false
BuildRoof@ProposeShouldNot = false

BuildWall@format = build the %s wall
BuildPillar@format = put up the %s pillar

ChoosePanel@format = find a panel to use
ChooseShard@format = find an ice shard to use

PickUpPanel@format = pick up a panel
PickUpShard@format = pick up an ice shard

PlacePanel@format = place the %s wall
PlaceShard@format = place the %s pillar
```

Translation File

```
let's_achieve_Shelter = all right, a clear path for once!

let's_not_achieve_BuildPillars = never mind about the pillars

we_should_find_a_panel_to_use = we need some walls
we_should_find_an_ice_shard_to_use = let's put some pillars at the front
```

Walrus Cave

The Walrus Cave level contains a third conversant: a "talking walrus" agent. Thus it was necessary to author the three task models listed below, one for each pair of actors.

Player and Walrus

```
<model about="urn:cetask.wpi.edu:models:secrets:WalrusCave"
       xmlns="http://www.cs.wpi.edu/~rich/tizona"
       xmlns:d="http://www.cs.wpi.edu/~rich/cetask/cea-2018-ext">

  <say id="Convo" actor="walrus" text="Ah, you have come at last. Welcome, great
emissaries, to my hall!">
    <say actor="player"
         text="Uh... hail, walrus! We seek the northern exit of your hall.">
      <say actor="walrus" text="What?! No one may exit the hall by the north gate
unless they know the secret of the Hall.">
        <say actor="player" text="Well, it\'s been great talking to you, great tusk,
but we really have to be going now.">
          <say actor="walrus" text="Not before you tell me the secret! I like the
two of you, so here\'s a clue:">
            <say actor="walrus" text="Living threats your senses hone">
              <say actor="walrus" text="The frozen twin smiling is seated">
                <say actor="walrus" text="There upon a warming throne">
                  <say actor="walrus" text="But you must not, or be defeated">
                    <say actor="walrus" text="Bring forth the adamantly polished">
                      <say actor="walrus" text="Hidden when walls were breached">
                        <say id="Secret" actor="walrus" text="Now, what is the secret?">
                          <say actor="player" text="A hidden door?">
                            <say actor="walrus" ref="Secret" text="I should say not!"/>
                          </say>
                          <say actor="player" text="A diamond?">
                            <say id="AdamantRequest" actor="walrus"
                                 text="Aha! The Adamant Request! Well done.">
                              <do task="OpenExit"/>
                            </say>
                          </say>
                          <say actor="player" text="A whale?">
                            <say actor="walrus" ref="Secret" text="Huh?"/>
                          </say>
                        </say>
                      </say>
                    </say>
                  </say>
                </say>
              </say>
            </say>
          </say>
        </say>
      </say>
    </say>
  </say>

  <d:task id="OpenExit">
    <d:binding slot="$this.external" value="false"/>
    <d:precondition> !exitOpen() </d:precondition>
    <d:postcondition sufficient="true"> exitOpen() </d:postcondition>
    <d:script>
      world.get("walrus").getLocation().setLocation(disappearLocation);
    </d:script>
  </d:task>

  <d:script init="true">
    function exitOpen() { return world.get("walrus").getLocation().x &lt; 0; }
    disappearLocation = new Packages.java.awt.Point(-2, -2);
  </d:script>

</model>
```

Player and Sidekick

```
<model about="urn:cetask.wpi.edu:models:secrets:WalrusCave"
       xmlns="http://www.cs.wpi.edu/~rich/tizona"
       xmlns:d="http://www.cs.wpi.edu/~rich/cetask/cea-2018-ext">

  <say actor="player" text="A whale?">
    <say actor="sidekick" text="That doesn\'t even make sense!"/>
  </say>

</model>
```

Sidekick and Walrus

```
<model about="urn:cetask.wpi.edu:models:secrets:WalrusCave"
       xmlns="http://www.cs.wpi.edu/~rich/tizona"
       xmlns:d="http://www.cs.wpi.edu/~rich/cetask/cea-2018-ext">
  <say id="Convo" actor="walrus"
       text="Ah, you have come at last. Welcome, great emissaries, to my hall!">
    <say actor="sidekick" text="Oh look, a talking walrus."/>
  </say>
  <say actor="walrus" text="What?! No one may exit the hall by the north gate unless
they know the secret of the Hall.">
    <say actor="sidekick"
        text="Seems to have a high opinion of himself for a talking pinniped."/>
  </say>
  <say id="Secret" actor="walrus" text="Now... what is the secret?">
    <say actor="sidekick" text="Finally. Thought he\'d never shut up."/>
  </say>
  <say id="AdamantRequest" actor="walrus" text="Aha! The Adamant Request! Well done.">
    <say actor="sidekick" text="He creeps me out. Let\'s get out of here."/>
  </say>
</model>
```

Appendix C

XML Schema for *Tizona* Format

The following listing is an XML Schema document that defines the form of *Tizona* format XML files. Such files can be transformed into standard ANSI/CEA-2018 task models using the XSL Transform file given in Appendix D.

```
<xs:schema targetNamespace="http://www.cs.wpi.edu/~rich/tizona"
    elementFormDefault="qualified"
    xmlns="http://www.cs.wpi.edu/~rich/tizona"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="model">
  <xs:complexType>
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="do"/>
        <xs:element ref="say"/>
      </xs:choice>
      <xs:any namespace="http://www.cs.wpi.edu/~rich/cetask/cea-2018-ext"
minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="about" type="xs:anyURI"/>
  </xs:complexType>
  </xs:element>

  <xs:element name="do">
    <xs:complexType>
      <xs:sequence>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="do"/>
          <xs:element ref="say"/>
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="optional"/>
      <xs:attribute name="actor" type="xs:string" use="optional"/>
      <xs:attribute name="task" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="say">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="applicable" type="xs:string" minOccurs="0" maxOccurs="1"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="do"/>
          <xs:element ref="say"/>
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="optional"/>
      <xs:attribute name="ref" type="xs:string" use="optional"/>
      <xs:attribute name="actor" type="xs:string" use="optional"/>
      <xs:attribute name="text" type="xs:string" use="optional"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Appendix D

Tizona to ANSI/CEA-2018 Translation

The following listing is an eXtensible Stylesheet Language (XSL) Transformation file that takes as input a *Tizona* format XML document and produces an ANSI/CEA-2018 task model.

```
<xsl:stylesheet version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.cs.wpi.edu/~rich/cetask/cea-2018-ext"
                xmlns:d="http://www.cs.wpi.edu/~rich/cetask/cea-2018-ext"
                xmlns:t="http://www.cs.wpi.edu/~rich/tizona">
  <xsl:output method="xml" indent="yes"/>
  <xsl:strip-space elements="*"/>
  <xsl:variable name="external" select="'player'"/>

  <xsl:template match="t:model">
    <xsl:element name="taskModel">

      <!-- force inclusion of disco namespace -->
      <xsl:namespace name="disco" select="'urn:disco.wpi.edu:Disco'" />

      <!-- take care of 'about' URN to make unique -->
      <xsl:attribute name="about">
        <xsl:value-of select="@about"/>
      </xsl:attribute>

      <!-- make top-level 'do' and 'say' tasks to get things started -->
      <xsl:for-each select="t:do|t:say">
        <xsl:element name="task">
          <xsl:attribute name="id">
            <xsl:value-of select="@id"/>
            <xsl:if test="not(@id)">
              <xsl:text>_</xsl:text>
              <xsl:value-of select="generate-id()"/>
            </xsl:if>
          </xsl:attribute>
          <xsl:apply-templates select="current()" mode="subtasks"/>
        </xsl:element>
      </xsl:for-each>

      <!-- make required 2nd, 3rd, and so on level tasks to build tree -->
      <xsl:apply-templates select="//t:do" mode="task"/>
      <xsl:apply-templates select="//t:say" mode="task"/>

      <!-- collect the disco elements -->
      <xsl:apply-templates select="d:task|d:script|d:subtasks" mode="disco"/>
    </xsl:element>
  </xsl:template>

  <!-- allow disco tags to pass through unscathed -->
  <xsl:template match="*" mode="disco">
    <xsl:element name="{local-name()}">
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates mode="disco"/>
    </xsl:element>
  </xsl:template>

  <!-- tasks for 'do' nodes -->
  <xsl:template match="t:do" mode="task">
    <!-- only generate task if there are children -->
    <xsl:if test="node()">
      <xsl:element name="task">
        <!-- use id attribute if present -->
        <xsl:attribute name="id">
          <xsl:choose>
            <xsl:when test="@id">
              <xsl:text>_</xsl:text>
              <xsl:value-of select="@id"/>
              <xsl:text>_tree</xsl:text>
            </xsl:when>
            <xsl:otherwise>
              <xsl:text>_</xsl:text>
              <xsl:value-of select="generate-id()"/>
              <xsl:text>_tree</xsl:text>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:attribute>

        <!-- apply to children -->
        <xsl:apply-templates select="t:do|t:say" mode="subtasks"/>
      </xsl:element>
    </xsl:if>
  </xsl:template>
```

```
<!-- tasks for 'say' nodes -->
<xsl:template match="t:say" mode="task">
  <!-- only generate task if there are children -->
  <xsl:if test="node()">
    <xsl:element name="task">
      <!-- use id attribute if present -->
      <xsl:attribute name="id">
        <xsl:choose>
          <xsl:when test="@id">
            <xsl:text>_</xsl:text>
            <xsl:value-of select="@id"/>
            <xsl:text>_tree</xsl:text>
          </xsl:when>
          <xsl:otherwise>
            <xsl:text>_</xsl:text>
            <xsl:value-of select="generate-id()"/>
            <xsl:text>_tree</xsl:text>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>

      <!-- apply to children -->
      <xsl:apply-templates select="t:do|t:say" mode="subtasks"/>
    </xsl:element>
  </xsl:if>
</xsl:template>

<!-- subtasks for 'do' nodes-->
<xsl:template match="t:do" mode="subtasks">
  <xsl:variable name="id">
    <xsl:choose>
      <xsl:when test="@id">
        <xsl:text>_</xsl:text>
        <xsl:value-of select="@id"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>_</xsl:text>
        <xsl:value-of select="generate-id()"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <xsl:element name="subtasks">
    <xsl:attribute name="id">
      <xsl:value-of select="$id"/>
      <xsl:text>_subtasks</xsl:text>
    </xsl:attribute>

    <!-- link to referenced task -->
    <xsl:element name="step">
      <xsl:attribute name="name">
```

```
        <xsl:value-of select="$id"/>
        <xsl:text>_step</xsl:text>
      </xsl:attribute>
      <xsl:attribute name="task">
        <xsl:value-of select="@task"/>
      </xsl:attribute>
    </xsl:element>

    <!-- include rest of tree if applicable -->
    <xsl:if test="node()">
      <xsl:element name="step">
        <xsl:attribute name="name">
          <xsl:value-of select="$id"/>
          <xsl:text>_ref</xsl:text>
        </xsl:attribute>
        <xsl:attribute name="task">
          <xsl:value-of select="$id"/>
          <xsl:text>_tree</xsl:text>
        </xsl:attribute>
      </xsl:element>
    </xsl:if>

    <xsl:apply-templates select="t:applicable" mode="disco"/>

    <!-- apply external modifiers if actor specified -->
    <xsl:if test="@actor">
      <xsl:element name="binding">
        <xsl:attribute name="slot">
          <xsl:text>$</xsl:text>
          <xsl:value-of select="$id"/>
          <xsl:text>s.external</xsl:text>
        </xsl:attribute>
        <xsl:attribute name="value">
          <xsl:if test="@actor=$external">
            <xsl:text>true</xsl:text>
          </xsl:if>
          <xsl:if test="@actor!=$external">
            <xsl:text>false</xsl:text>
          </xsl:if>
        </xsl:attribute>
      </xsl:element>
    </xsl:if>
  </xsl:element>
</xsl:template>

<!-- subtasks for 'say' nodes -->
<xsl:template match="t:say" mode="subtasks">
  <xsl:variable name="id">
    <xsl:choose>
      <xsl:when test="@id">
        <xsl:text>_</xsl:text>
        <xsl:value-of select="@id"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>_</xsl:text>
        <xsl:value-of select="generate-id()"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <xsl:element name="subtasks">
    <xsl:attribute name="id">
      <xsl:value-of select="$id"/>
      <xsl:text>_subtasks</xsl:text>
    </xsl:attribute>

    <xsl:if test="@ref">
      <!-- referencing another node -->
      <xsl:element name="step">
        <xsl:attribute name="name">
          <xsl:value-of select="$id"/>
          <xsl:text>_step</xsl:text>
        </xsl:attribute>
        <xsl:attribute name="task">
          <xsl:text>_</xsl:text>
          <xsl:value-of select="@ref"/>
          <xsl:text>_tree</xsl:text>
        </xsl:attribute>
      </xsl:element>
    </xsl:if>

    <xsl:if test="not(@ref)">
      <xsl:element name="step">
        <xsl:attribute name="name">
          <xsl:value-of select="$id"/>
          <xsl:text>_step</xsl:text>
        </xsl:attribute>
        <xsl:attribute name="task">
          <xsl:text>disco:edu.wpi.disco.lang.Say</xsl:text>
        </xsl:attribute>
      </xsl:element>
    </xsl:if>

    <!-- include rest of tree if applicable -->
    <xsl:if test="node()">
      <xsl:element name="step">
        <xsl:attribute name="name">
          <xsl:value-of select="$id"/>
          <xsl:text>_ref</xsl:text>
        </xsl:attribute>
        <xsl:attribute name="task">
          <xsl:value-of select="$id"/>
          <xsl:text>_tree</xsl:text>
        </xsl:attribute>
```

```
        </xsl:element>
      </xsl:if>

      <xsl:apply-templates select="t:applicable" mode="disco"/>

      <xsl:if test="not(@ref)">
        <!-- include text -->
        <xsl:element name="binding">
          <xsl:attribute name="slot">
            <xsl:text>$</xsl:text>
            <xsl:value-of select="$id"/>
            <xsl:text>_step.text</xsl:text>
          </xsl:attribute>
          <xsl:attribute name="value">
            <xsl:text>'</xsl:text>
            <xsl:value-of select="@text"/>
            <xsl:text>'</xsl:text>
          </xsl:attribute>
        </xsl:element>
      </xsl:if>

      <!-- apply external modifiers if actor specified -->
      <xsl:if test="@actor">
        <xsl:element name="binding">
          <xsl:attribute name="slot">
            <xsl:text>$</xsl:text>
            <xsl:value-of select="$id"/>
            <xsl:text>_step.external</xsl:text>
          </xsl:attribute>
          <xsl:attribute name="value">
            <xsl:if test="@actor=$external">
              <xsl:text>true</xsl:text>
            </xsl:if>
            <xsl:if test="@actor!=$external">
              <xsl:text>false</xsl:text>
            </xsl:if>
          </xsl:attribute>
        </xsl:element>
      </xsl:if>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

Bibliography

[1] Adams, E. *Fundamentals of Game Design.* New Riders, 2009.

[2] Cavazza, M., and Charles, F. Dialogue generation in character-based interactive storytelling. *AAAI First Annual Artificial Intelligence and Interactive Digital Entertainment Conference* (2005), 12–17.

[3] Charles, F., Mead, S., and Cavazza, M. Generating dynamic storylines through characters interactions. *International Journal of Intelligent Games & Simulation*, 1 (2002), 5–11.

[4] Consumer Electronics Association. *CEA-2018 Task Model Description.* http://ce.org/cea-2018, 2007.

[5] Despain, W. *Writing for Video Game Genres: From FPS to RPG.* A K Peters, 2008.

[6] Golden T Studios. http://goldenstudios.or.id/.

[7] Heineman, G., Pollice, G., and Selkow, S. *Algorithms in a Nutshell.* O'Reilly Media, 2008.

[8] Isla, D. Handling complexity in the Halo 2 AI. *Gamasutra Online* (2005), http://www.gamasutra.com/gdc2005/features/20050311/isla_pfv.htm.

[9] Kelly, J., Botea, A., and Koenig, S. Offline planning with hierarchical task networks in video games. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, Stanford, CA* (2008).

[10] Lebowitz, M. Story-telling as planning and learning. *Poetics 14*, 6 (1985).

[11] Lesh, N., Rich, C., and Sidner, C. Using plan recognition in human-computer collaboration. 23–32.

[12] Mateas, M., and Stern, A. Façade: An experiment in building a fully-realized interactive drama. In *Game Developers Conference, Game Design Track* (2003), http://www.interactivestory.net/papers/MateasSternGDC03.pdf.

[13] McCannel, D., and Ivory, C. Forerunner project. Major qualifying project, Worcester Polytechnic Institute, 2008.

[14] Oracle/Collabnet. Liveconnect support in the new java plug-in technology, 2006.

[15] Owen, C., Biocca, F., Bohil, C., and Conley, J. SIMDIALOG: A visual game dialog editor. *Proceedings of Meaningful Play Conference, Michigan State University, East Lansing, MI* (2008).

[16] Rich, C. Building task-based user interfaces with ANSI/CEA-2108. *IEEE Computer 42*, 8 (2009), 20–27.

[17] Rich, C., and Sidner, C. Collagen: A collaboration manager for software interface agents. *User Modeling and User-Adapted Interaction 8* (1998), 315–350.

[18] Schach, S. R. *Object Oriented and Classical Software Engineering, 6th Ed.* McGraw-Hill, 2005.

[19] Sidner, C. An artificial discourse language for collaborative negotiation. *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994), 814–819.

[20] Skorupski, J., Jayapalan, L., Marquez, S., and Mateas, M. Wide Ruled: A friendly interface to author-goal based story generation. *Lecture Notes in Computer Science 4871* (2007), 26–38.

[21] Sommerville, I. *Software Engineering, 7th Ed.* Pearson Education Ltd., 2004.

[22] Winston, P. *Artificial Intelligence.* Addison-Wesley, 1992.