University of Alberta

CODE OPTIMIZATION AND DETECTION OF SCRIPT CONFLICTS IN VIDEO GAMES

by

Yi YANG

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©Yi YANG
Fall 2009
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Examining Committee

Duane Szafron, Computing Science

Michael Carbonaro, Educational Psychology

José Nelson Amaral, Computing Science

Abstract

Scripting languages have gained popularity in video games for specifying the interactive content in a story. Game designers do not necessarily possess programming skills and often demand code-generating tools that can transform textual or graphical descriptions of interactions into scripts interpreted by the game engine. However, in event-based games, this code generation process may lead to potential inefficiencies and conflicts if there are multiple independent sources generating scripts for the same event. This thesis presents solutions to both perils: transformations to eliminate redundancies in the generated scripts and an advisory tool to provide assistance in detecting unintended conflicts. By incorporating traditional compiler techniques with an original code-redundancy-elimination approach, the code transformation is able to reduce code size by 25% on scripts and 14% on compiled byte-codes. With the proposed alternative view, the advisory tool is suitable for offering aid to expose potential script conflicts.

Acknowledgement

I would like to thank my supervisor Duane Szafron, under whose guidance I completed this research. I would also like to thank all my fellow students in the ScriptEase group: Neesha Desai, Christopher Kerr, Marcus Trenton and Richard Zhao, who provided valuable feedback throughout my research. Special thanks to the ScriptEase implementation team members – Jason Duncan, Josh Friesen and Robin Miller, as well as former project participants – Ana Alcantara and Jeff Siegel.

Table of Contents

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Event-based Code Generation . . . 1
- 1.3 Potential Inefficiencies . . . 2
- 1.4 Potential Conflicts . . . 2
- 1.5 ScriptEase . . . 4
- 1.6 ScriptEase Pattern Examples . . . 5
- 1.7 Inefficiencies in ScriptEase . . . 10
- 1.8 Conflicts in ScriptEase/*NWN* . . . 12
- 1.9 Summary . . . 14

2 Optimization **15**
- 2.1 Related Work . . . 15
  - 2.1.1 Criteria for Code-improving Transformations . . . 15
  - 2.1.2 Peephole Optimization . . . 16
  - 2.1.3 Global Optimization . . . 17
  - 2.1.4 Optimizing Scripting Languages . . . 19
- 2.2 Optimizing ScriptEase-generated NWScript . . . 21
- 2.3 Optimization Techniques Used . . . 21
  - 2.3.1 Constant Folding . . . 21
  - 2.3.2 Dead-branch Elimination . . . 22
  - 2.3.3 Function Inlining . . . 22
  - 2.3.4 Redundant Code Elimination . . . 27
- 2.4 Script Transformation Implementation . . . 32
- 2.5 Evaluation . . . 33
- 2.6 Summary . . . 35

3 Conflict Advisor **36**
- 3.1 Related Work . . . 36
  - 3.1.1 ScriptEase Pattern Builder . . . 36
  - 3.1.2 ScriptEase Code Viewer . . . 38
- 3.2 Conflict Advisor with Object-slot View . . . 40
  - 3.2.1 A Tree Representation . . . 40
  - 3.2.2 The Object-slot View . . . 41
  - 3.2.3 Pattern Builder vs. Object-slot View . . . 43
  - 3.2.4 Filtering Game Objects . . . 44
- 3.3 Summary . . . 45

4 Conclusion and Future Work **47**

Bibliography **49**

A Code from *The Summer Story – Phase 1* **51**

List of Tables

2.1 Modules used to evaluate the transformations 33
2.2 Evaluation of code transformation on ScriptEase-scripted NWN Modules 34

List of Figures

1.1 A screenshot showing the designer setting parameter of a pattern in ScriptEase . . . 5
1.2 An encounter pattern in ScriptEase 6
1.3 A quest pattern in ScriptEase 7
1.4 A dialogue pattern in ScriptEase 8
1.5 A behaviour pattern in ScriptEase 9
1.6 A potential conflict example in ScriptEase 13

2.1 Example of constant folding 22
2.2 Architecture of the code transformer 32

3.1 ScriptEase Pattern Builder 37
3.2 Folders in ScriptEase pattern builder 37
3.3 ”View Code” in ScriptEase 39
3.4 ScriptEase Code Viewer 39
3.5 Tree view in the Pattern Builder 40
3.6 Tree view in the Object-slot View 41
3.7 Object-slot tree of the “Kill Dragon” quest 42
3.8 The Object-slot View in ScriptEase 43
3.9 The Object-slot View with “show conflict only” turned on 44
3.10 Object Filter for Conflict Advisor 45
3.11 ScriptEase Conflict Advisor 46

Chapter 1

Introduction

1.1 Overview

Interactive story-telling is one of the major features in video games, especially role-playing games. In recent games, the story, including character behaviours, quests and dialogues, etc., is usually specified by scripting languages or finite state machines. Scripting languages (e.g. Python [26] and Lua [18]) are used because they are simpler than programming languages (e.g. C/C++, Java) for game designers, who are often non-programmers, but powerful enough to accomplish the task. More importantly, script code does not need to be compiled for execution, making it ideal for the data-driven design of video games, in which a compiled game engine processes core logic and graphics while script code serves as modifiable content.

However, scripting languages are still not simple enough for non-programmers as coding in these languages requires at least some programming experience. Therefore code-generating tools would improve the process. From a game designer's perspective, there is a strong wish to design each particular piece of game content (e.g. a quest, a character behaviour) at a high level. This can be accomplished if the script code is generated behind the scenes from high-level descriptions. A number of such code-generating tools have been made available for various commercial video games (e.g. World Editor for *WarCraft III* [30], ScriptEase for *Neverwinter Nights* [22]), which, to some extent, fulfill this need.

1.2 Event-based Code Generation

The event-based model is common in video game design and follows a "when an event happens, perform some actions" paradigm. The game engine checks when one of the pre-defined events on a particular game object is fired and executes the script attached to that event-slot by performing the actions described in the script code. For instance, in the game *Neverwinter Nights* if a game designer wishes to include the scenario "when the player character (PC) pulls a lever, spawn a creature" in the story, a piece of script code that implements "creature spawning" needs to be written and attached to the lever's *OnUsed* slot. In the game, when the PC pulls the lever, the game engine is aware that

the *OnUsed* event is fired and executes the script that is attached to this slot, which performs the *spawning a creature* action.

The challenge of event-based code generation is to combine code generated from independent sources. It is very common that some game objects are involved in several scenarios of the story. For instance, a particular creature may be involved in a quest that requires the PC to kill it and another quest that drops a plot item when it dies. Because those two quests do not depend on each other (the game designer can remove either one without affecting the other), two code snippets need to be generated independently, but both are attached to the same *OnDeath* slot of that creature. One snippet causes a journal entry to pop up in the quest, and the other causes a plot item to appear. Since each event-slot has one single script, code generated from independent sources has to be integrated into this single script while retaining the syntactic and semantic correctness of both sources.

The most straightforward approach to code generation is for each game component (quest, character behaviour, etc.) to generate its own code (usually in the form of functions) and to combine them afterwards in a single script that calls these functions in sequence. However, this naive code combination process may create perils such as potential inefficiencies and conflicts.

1.3 Potential Inefficiencies

Despite the fact that event-based code generation comes from independent sources, the code snippets still share the same event on a common game object, which may result in substantial similarities among these generated snippets. For instance, in a typical script attached to an item's *OnUsed* slot, common information such as who used this item, the name of this item, the way it was used (e.g. if a door was opened or closed) may be used by multiple code snippets. If multiple generators are generating code for this particular event, each generator may independently create calls to the game engine to gather this common information. When independent functions are called in sequence, there will be multiple lines of identical and redundant code in these functions. The redundant code causes execution inefficiency. It is my hypothesis that this inefficiency can be reduced or eliminated.

1.4 Potential Conflicts

Another peril of generating event-based code from independent generators is the potential for code interactions to lead to unintended consequences (which I call conflicts). Many kinds of conflicts could occur of which I present three here.

First, the sequence of functions from independent generators has an arbitrary order. If the order does not matter there is no problem. However, if it does matter, the game designer must be made aware of it. In other words, from the designer's perspective they may be independent scenarios, while in reality the functions are called sequentially in the order they're generated, so interactions can occur.

For example, assume a game designer makes two independent scenarios: 1. When the PC pulls a lever, open the door nearest to the lever; 2. When the PC pulls a lever, spawn a creature and destroy the lever. These two scenarios are intended to be independent. The first scenario is a general pattern attached to almost every lever in the game, while the second is part of this specific story. The author's intention is that the spawned creature will attack the PC and the lever will be destroyed, depriving the PC of the ability to toggle the door. The author wants to prevent the PC from separating the creature and itself using the door. However, the door must open when the lever is pulled, otherwise there would be no way out of this room. Code snippets generated from both scenarios are inserted into the same script attached to the *OnUsed* slot of the lever. On one hand, the designer might not be aware of the order of these two scenarios and assumes they occur simultaneously; on the other hand, the designer might be aware of the order and intends the 1st scenario to occur before the 2nd, falsely assuming that code would be generated in the order the scenario is constructed. In either case, the designer would see no ordering problem. In fact, the code snippet of the 2nd scenario could precede that of the 1st when their functions are called in the one single script. If the code is generated in this order, then pulling the lever will destroy it so there is no way for the game engine to compute the door nearest to the now-destroyed lever afterwards.

Second, code from independent sources may have order-independent side-effects (e.g. changing the status of an object) that cause conflicts. For example, assume a game designer puts several doors and levers in a large room and makes default scenarios for every door-lever pair: when the PC pulls a lever, toggle the corresponding door from open to closed or from closed to open. Then, assume another designer uses one specific door in that room. This designer creates a scenario: when the PC pulls the lever for that door, a number of actions will occur : 1. toggle the door; 2. spawn a creature; 3. create a visual effect. Since both designers work on the scenario level without touching the scripts, they have no idea what is already in any script attached to that door. In a large project, they would not have time to go through every scenario made that might affect the door. In this case, no matter what order the code is generated, the consequence will be that when the PC pulls the lever, the door will not be toggled as its open or closed binary state has been changed twice.

Third, dynamic conflicts could occur at run-time. For example, assume a designer makes two scenarios: 1. When the PC steps on a trigger, a fireball is fired at the PC; 2. When the PC steps on a trigger, move a statue to a door and make it open the door. If the designer uses the same trigger for both scenarios, code generated from both scenarios will be inserted into the same script. There does not appear to be any conflict between these scenarios at design time (or "compile time" if we consider the code generator a compiler). However, during the game the fireball does AOE (area of effect) damage so it may destroy the statue while hitting the PC near it. As a result, the destroyed statue could not move and open the door. This type of conflict is very difficult to detect since the first scenario does not explicitly refer to the statue.

1.5 ScriptEase

ScriptEase [28, 19] is a software tool that helps game authors create content without coding any scripts. The goal is to aid authors in the creation of content from high-level descriptions (patterns) while code is generated behind the scenes. This goal is fulfilled by using generative design patterns - patterns that can automatically be translated into executable code [5, 10]. In ScriptEase, creating game content is simplified to a three-step process [8]:

1. Select an appropriate pattern and create an instance of it,
2. Adapt the pattern description,
3. Click a button and scripting code is automatically generated.

The automatically-generated scripting code is executed by the game engine during game-play. However sometimes, technical designers may modify the generated code for customized situations. In addition, the ScriptEase team often looks at the generated code when debugging ScriptEase. Therefore, the generated code should be readable by humans.

A pattern in ScriptEase is a representation of an interactive scenario in games (e.g. an encounter between the PC and a game object, a quest, a character behaviour). ScriptEase provides a rich pattern catalogue for role-playing games [23]. For example, when a lever is pulled, the author might want to have a nearby door toggled. This can be done using the pattern *placeable used - toggle door*. The author then adapts the description by specifying the placeable (the lever) and the door to be toggled. When the author presses a button, the scripting code is generated automatically from the adapted pattern. Figure 1.1 shows a screenshot of the *placeable used - toggle door* pattern in ScriptEase, in which the designer is setting the placeable to a specific lever.

ScriptEase currently supports four categories of patterns:

1. Encounters – simple character/object interactions. For example, when the PC enters a trigger, a creature may be spawned.
2. Quests – a set of objectives that the PC needs to accomplish in order to advance the storyline. For example, when the PC kills the dragon, the village is saved and the PC gets rewards.
3. Dialogues – conversations between the PC and non-player characters (NPCs) are dynamic throughout the story. For example, an NPC may speak different lines to the PC before and after the first time they meet.
4. Behaviours – besides the PC, there are many NPCs populating the game world. Each NPC needs to have specified behaviours to look natural and intelligent. For example, an NPC can be assigned a *guard* behaviour to make it secures a chest against thieves.

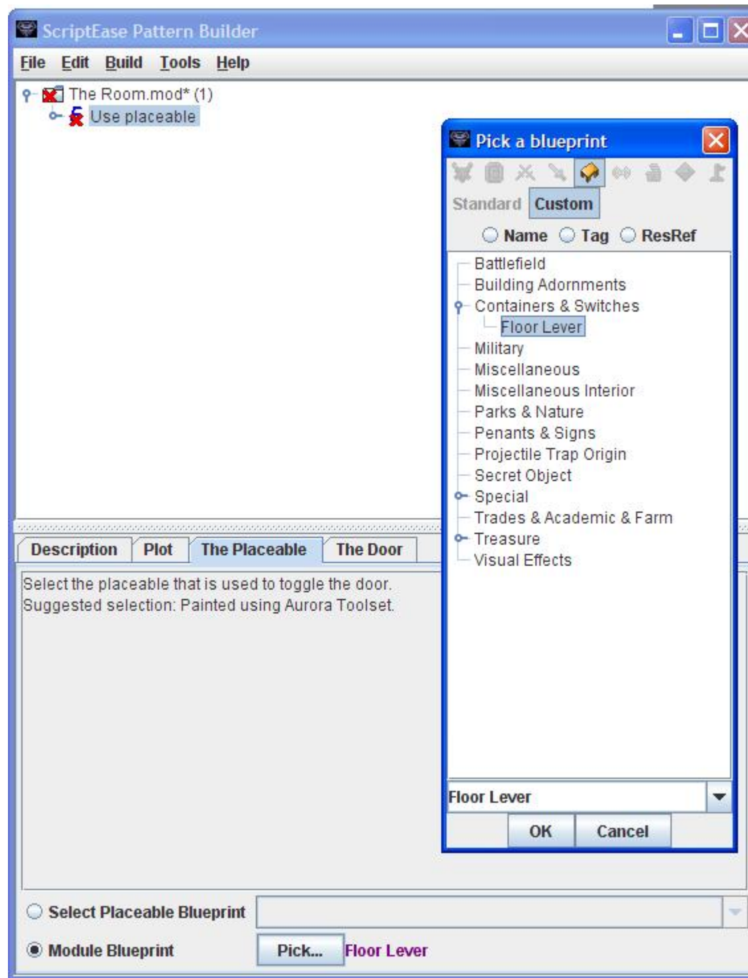

Figure 1.1: A screenshot showing the designer setting parameter of a pattern in ScriptEase

In this dissertation, I study code generated for BioWare's [3] game *Neverwinter Nights (NWN)*. The scripting language used to write stories for this game is called NWScript, a C-like procedural language. NWScript is event-based; ScriptEase generates NWScript code from each kind of pattern (encounters, quests, dialogues and behaviours). This is an appropriate example of event-based code generation from independent sources, which is the focus of this dissertation.

1.6 ScriptEase Pattern Examples

Each kind of ScriptEase pattern generates code on an event-slot. The generated code is based on the object and its event-slot. To understand the perils of independent code generation, it is necessary to appreciate the diversity of ScriptEase pattern components.

1. Example of an encounter pattern

   Figure 1.2 shows an encounter pattern: *Trigger enter - barrier*. The selected "E" block and all the blocks inside (highlighted in a red box) represent an instance of the *Trigger enter - barrier*

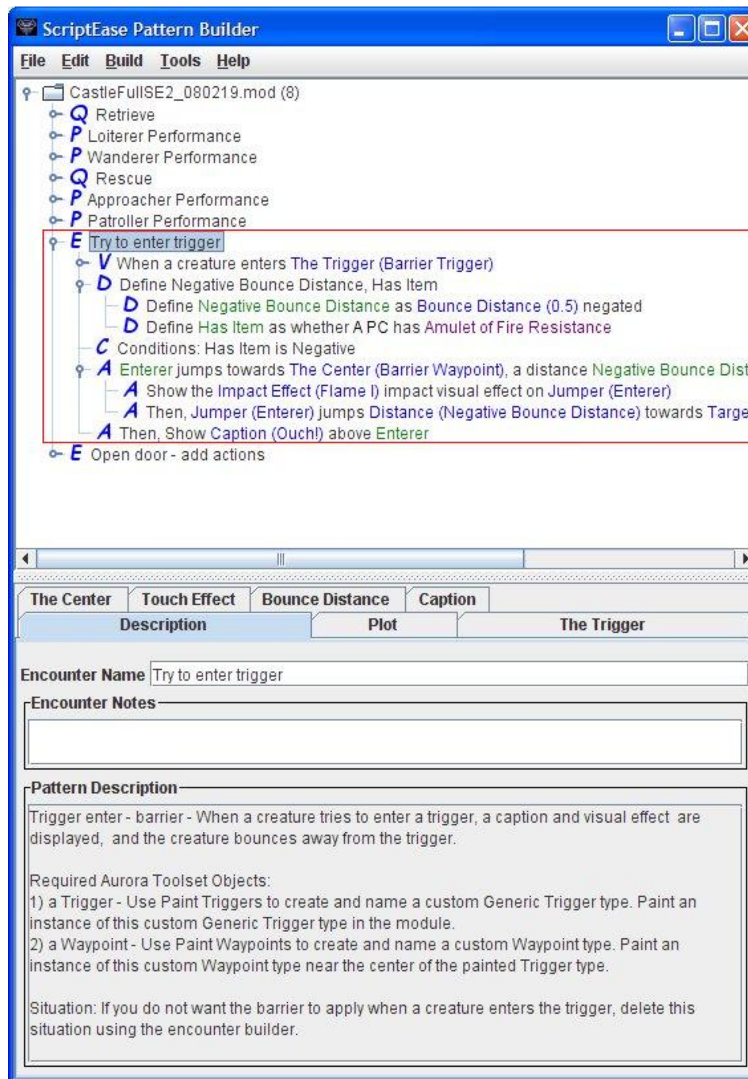

Figure 1.2: An encounter pattern in ScriptEase

encounter pattern, in which "E" represents an *encounter*. An encounter pattern consists of several components:

- "V" represents an *event*, the event to which this encounter is attached; "V" determines where the generated code is inserted.
- "D" represents a *definitions* (or a *definition block* – a list of definitions), which defines the value of a variable used in this scope. A definition usually generates a variable declaration and an assignment in NWScript.
- "A" represents an *actions* (or an *action block* – a list of actions), which is a list of calls to the game engine. The code generated from an action is usually a sequence of NWScript API[1] calls.

[1] Pre-defined NWScript functions that asks the game engine to perform a task or retrieve a query.

- "C" represents a *condition*, which guards the actions following it. These actions will only be performed if the condition is true. Conditions generate if-statements in NWScript.

2. Example of a quest pattern

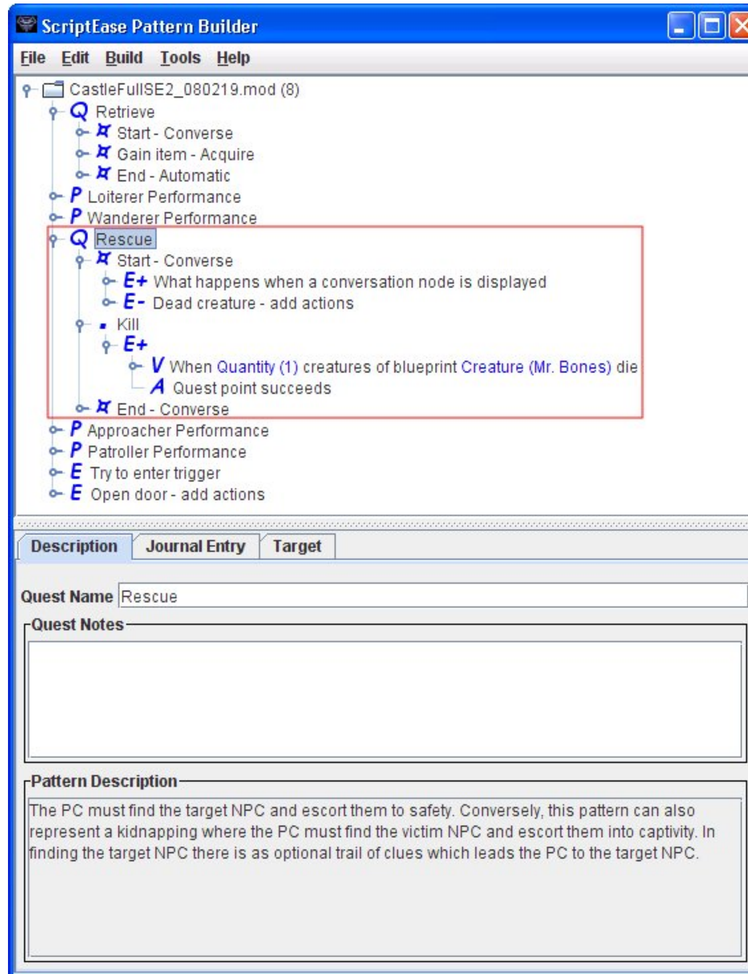

Figure 1.3: A quest pattern in ScriptEase

Figure 1.3 shows a quest pattern: *Rescue*. The selected "Q" block and all the blocks inside (highlighted in a red box) represent an instance of the *Rescue* quest pattern, in which "Q" represents a *quest*. The components of a quest pattern are:

- The dot symbol (before "Kill") represents a *quest point* – a milestone in a quest and is usually associated with journal entries in the game.
- The star symbol (before "Start - Converse") represents a *meta quest point* – a template for quest points that can be replaced with specific *quest points*.
- "E+" inside a quest point (or meta quest point) represents a *success encounter* – a regular encounter pattern instance with a *success action* that enables the PC to complete this quest point if this encounter is activated.

- “E-” inside a quest point (or meta quest point) represents a *fail encounter* – a regular encounter pattern instance with a *fail action* that enables the PC to fail this quest point if this encounter is activated. “E+” and “E-” both contain “V”, “D”, “A” and “C” as described in encounter patterns.

Code generation for quests is still event-based, meaning that only “E+” and “E-” generate code because they are attached to events. Since a quest may have multiple quest points and meta quest points, it potentially generates code at different event destinations and may interact with code generated by encounters or other quests in the same module.

3. Example of a dialogue pattern

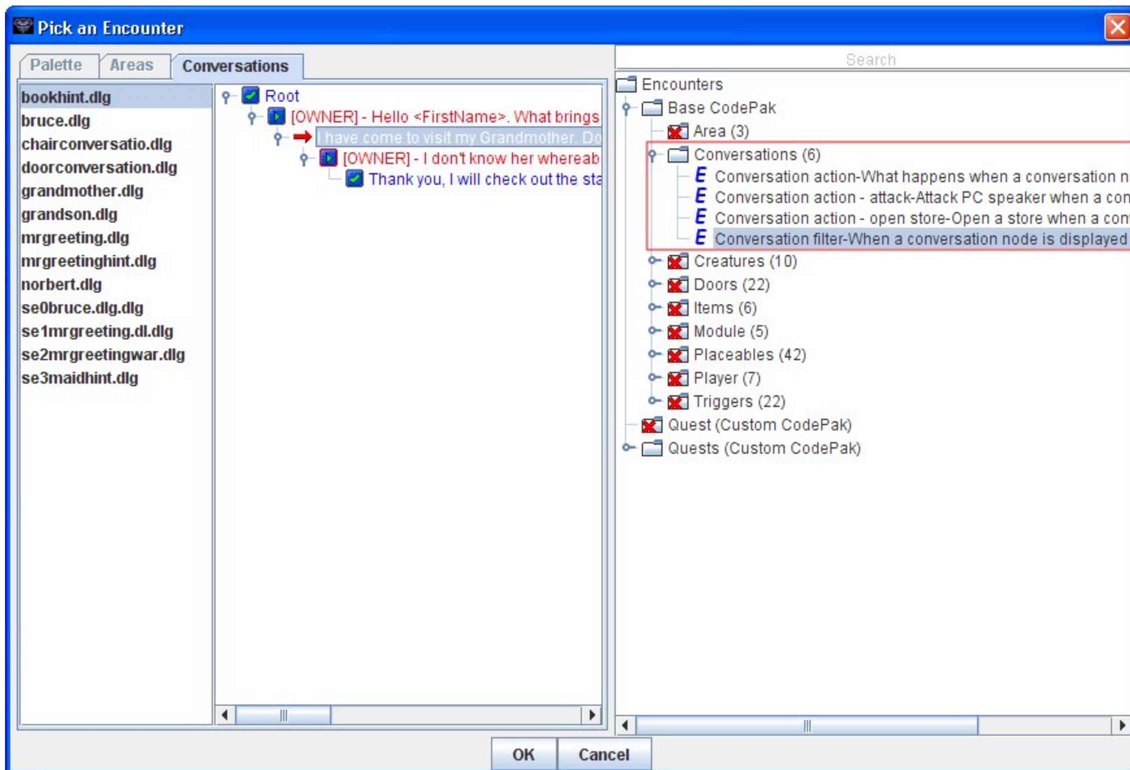

Figure 1.4: A dialogue pattern in ScriptEase

In the current ScriptEase version, dialogue patterns are implemented as encounter patterns and they have the same components. Figure 1.4 shows how to instantiate a dialogue pattern. The user needs to pick an existing conversation node in the module in order to create a dialogue pattern instance. There are currently two types of dialogue patterns in ScriptEase: *Conversation Action* and *Conversation Filter*. A *Conversation Action* pattern generates code for the *Script* event and a *Conversation Filter* pattern generates code for the *Active* event, both on a conversation node.

4. Example of a behaviour pattern

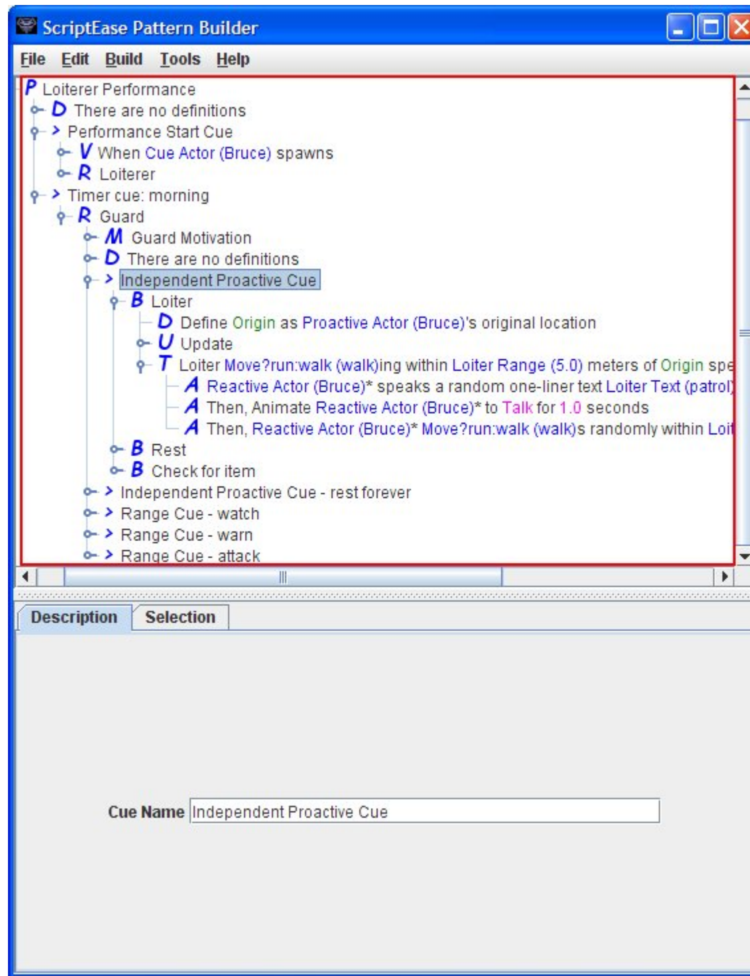

Figure 1.5: A behaviour pattern in ScriptEase

Figure 1.5 shows a behaviour pattern in ScriptEase. The selected “P” block and all the blocks inside (highlighted in a red box) represent an instance of the *Loiterer performance* (“P”). A performance is the highest-level component of a generic behaviour patter. Unlike the other three kinds of patterns, behaviour patterns are not named by the highest-level component, performance but rather, by the most important sub-component of a performance, behaviour. The components used with behaviour patterns are:

- “>” represents a *cue*. There are two kinds of cues: role cues, which pick roles in a performance, and behaviour cues, which pick behaviours in roles. Each cue is triggered by an event.
- “R” represents a *role*, which consists of a set of related cues, where each cue contains the hehaviours it can trigger.
- “M” represents a *motivation*, which determines how a cue selects between its behaviours.
- “B” represents a *behaviour*, which is a set of tasks.

- “T” represents a *task*, which is a set of actions.
- “U” represents an *update*, which updates a motivation after its behaviour has been performed.

Code generation for behaviour patterns is complicated. Cues determine the event-slot to which the generated code is attached. In general, a cue can generate a script on any slot of any object. For example, the *Performance Start Cue* in the performance generates code on the *OnSpawn* slot of the creature to which the performance is attached, while the *Independent Proactive Cue* in the *Guard* role generates code on the *OnUserDefined* slot of the creature. Since the generated code may be attached to any slot of any object, this code may interact with that generated by encounter, quest and dialogue patterns. This further complicates the issue discussed in this dissertation: event-based code generation from independent sources could cause potential inefficiencies and conflicts.

1.7 Inefficiencies in ScriptEase

Since ScriptEase independently generates scripts from different patterns and then combines them if they are on the same event of an object, their generators do not know if there are other scripts attached to the same event until this generation process is finished. Therefore, the generated code may have many redundancies, such as duplicate API calls.

For example, assume a game designer creates two encounter patterns *placeable use - toggle door* and *placeable use - spawn creature*, and sets the *placeable* to the same lever object. ScriptEase generates two code snippets, combines them into one single script and attaches it to the *OnUsed* slot of the lever. The generated code is shown in Code 1.1.

```
1  void Useplaceable_0() {
2      // 1st pattern: placeable use - toggle door
3      // (code omitted)
4  }
5  void Useplaceable_1() {
6      // 2nd pattern: placeable use - spawn creatures
7      // (code omitted)
8  }
9  void main() {
10     // Calls the function for the 1st pattern
11     Useplaceable_0();
12
13     // Calls the function for the 2nd pattern
14     Useplaceable_1();
15 }
```

Code 1.1: Generated skeleton code

When the game designer presses the button that automatically generates code, ScriptEase compiles the pattern *placeable use - toggle door* to a function called `Useplaceable_0()` and pattern

placeable use - spawn creature to `Useplaceable_1()`. Then these two functions are inserted into one single script that contains `main()`, which is the execution entry point of this script. When an event fires, the game engine executes the `main()` function in the script. Finally, ScriptEase generates calls to `Useplaceable_0()` and `Useplaceable_1()` in `main()`.

Despite the fact that the two functions `Useplaceable_0()` and `Useplaceable_1()` perform different tasks, they have a lot in common in their function body, because they are generated for the same event of a unique game object.

Code 1.2 shows the complete code for these two functions generated by ScriptEase. Identical lines of code are marked in bold.

```
1  void Useplaceable_0() {
2
3      // The following are all of the variables used in this situation
4      int OpenorClose_SE5;
5      object ThePlaceable_SE1;
6      int UnlockorLock_SE4;
7      object User_SE0;
8      object StoneDoor_SE2;
9      int DoorLocked_SE3;
10
11     // When Stone Door Lever is used
12     if( !  TRUE ) return;
13
14     // Get the object with tag "UA_StoneDoor"
15     StoneDoor_SE2 = GetNearestObjectByTag("UA_StoneDoor");
16     if (StoneDoor_SE2 == OBJECT_INVALID )
17     StoneDoor_SE2 = GetObjectByTag("UA_StoneDoor");
18
19     // Define Door Locked as whether Stone Door is locked
20     DoorLocked_SE3 = GetLocked(StoneDoor_SE2);
21
22     // Define Unlock or Lock as the opposite of Door Locked
23     UnlockorLock_SE4 = ! DoorLocked_SE3;
24
25     // Define Open or Close as the opposite of Unlock or Lock
26     OpenorClose_SE5 = ! UnlockorLock_SE4;
27
28     // Define User as the user of Stone Door Lever
29     User_SE0 = GetLastUsedBy();
30
31     // Define The Placeable as the object the event is fired on
32     ThePlaceable_SE1 = OBJECT_SELF;
33
34     // Define Door Locked as whether Stone Door is locked
35     DoorLocked_SE3 = GetLocked(StoneDoor_SE2);
36
37     // Define Unlock or Lock as the opposite of Door Locked
38     UnlockorLock_SE4 = ! DoorLocked_SE3;
39
40     // Define Open or Close as the opposite of Unlock or Lock
```

```
41      OpenorClose_SE5 = ! UnlockorLock_SE4;
42
43      // Main code body - checks conditions and executes actions
44      // Unlock or Lock Stone Door
45      SetLocked(StoneDoor_SE2, UnlockorLock_SE4);
46
47      // Stone Door* Open or Closes Stone Door
48      SE_Ac_OpenCloseDoorTodo(StoneDoor_SE2, StoneDoor_SE2,
49          OpenorClose_SE5);
50  }
51  void Useplaceable_1() {
52
53      // The following are all of the variables used in this situation
54      object SpawnedCreature_SE2;
55      object ThePlaceable_SE1;
56      object User_SE0;
57
58      // When Stone Door Lever is used
59      if( !  TRUE ) return;
60
61      // Define User as the user of Stone Door Lever
62      User_SE0 = GetLastUsedBy();
63
64      // Define The Placeable as the object the event is fired on
65      ThePlaceable_SE1 = OBJECT_SELF;
66
67      // Main code body - checks conditions and executes actions
68      // Spawn Spawned Creature from Evil Penguin near The Placeable
69      SpawnedCreature_SE2 = SE_Ac_SpawnCreatureNearObject(
70          "penguin002",   ThePlaceable_SE1);
71
72      // Show the Unsummon impact visual effect on Spawned Creature
73      SE_Ac_ShowImpactVisualEffectOnCreature(VFX_IMP_UNSUMMON,
74          SpawnedCreature_SE2);
75  }
```

Code 1.2: Complete function body

The redundant lines of code include variable declarations (`object User_SE0`), variable assignments (`ThePlaceable_SE1 = OBJECT_SELF`), NWScript API calls (`User_SE0 = GetLastUsedBy()`) and other expressions (`if(! TRUE)return`), all of which can be optimized. One of the main research contributions in this thesis is a collection of optimization techniques that eliminate these redundancies.

1.8 Conflicts in ScriptEase/*NWN*

Figure 1.6 shows how the first example discussed in Section 1.4 looks like in ScriptEase. In this example, there are two encounter patterns in the module (there could be more to complicate this problem): *Pull lever - spawn creature and destroy lever* and *Pull lever - open the nearest door*. They are attached to the *OnUsed* slot of the same lever object. Therefore, code generated from

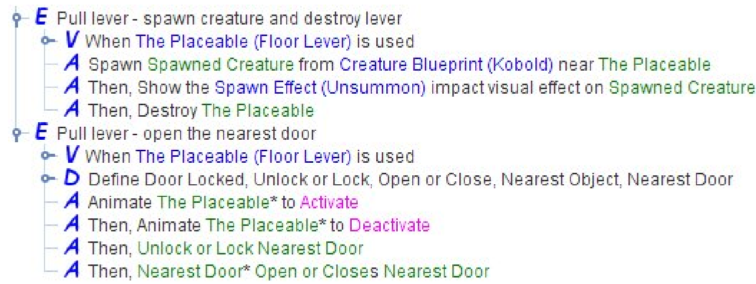

Figure 1.6: A potential conflict example in ScriptEase

these two patterns are inserted into one script, which is partially shown in Code 1.3. The two bold lines mark the conflict. As shown in the code, Pattern *Pull lever - spawn creature and destroy lever* generates code before Pattern *Pull lever - open the nearest door*, so that the `DestroyObject()` call precedes the `GetNearestObject()` call. Therefore the latter code is unable to access the destroyed object, `ThePlaceable_SE1`.

```
1   #include "i_se_aux"
2   /*-- Code omitted: Function declarations --*/
3
4   void PlaceableusespawncreatureUseplaceable_1() {
5
6     /*-- Code omitted --*/
7
8     // Destroy The Placeable
9     DestroyObject(ThePlaceable_SE1);
10  }
11
12  void PlaceableusetoggledoorUseplaceable_2() {
13
14    /*-- Code omitted --*/
15
16    // Define Nearest Object as the nearest object to The Placeable with kind Door
17    NearestObject_SE6 =
18          GetNearestObject(OBJECT_TYPE_DOOR, ThePlaceable_SE1);
19
20    /*-- Code omitted --*/
21  }
22
23  void main() {
24
25    PlaceableusespawncreatureUseplaceable_1();
26    PlaceableusetoggledoorUseplaceable_2();
27
28  }
```

Code 1.3: Generated code with potential conflict

Since ScriptEase has a pattern-based GUI and currently supports four different kinds of patterns (encounter, quest, behaviour and dialogue) it is difficult for game designers to detect the potential

conflicts such as the one shown in Figure 1.6. Another research contribution of this thesis is a new ScriptEase view that aids designers in detecting these conflicts.

1.9 Summary

This chapter first briefly discussed the use of scripting languages in video games and game designers' demand for tools that automatically generate scripts. The event-based model for video-games scripts was then introduced. This chapter explained, for this model, how code is generated from independent sources to the specific event-slots. Based on properties of this mechanism, I identified potential inefficiencies and conflicts that could be found in the generated scripts. In addition, a specific code-generating tool for video games – ScriptEase was highlighted, with examples of potential inefficiencies and conflicts in ScriptEase-generated code.

This thesis makes two contributions: it shows how to eliminate the potential inefficiencies by performing transformations on ScriptEase-generated code (Chapter 2) and it assists game designers to identify potential conflicts by providing an alternative view in ScriptEase (Chapter 3). Chapter 2 will first look at the related work on optimizing scripting languages, describe the optimization techniques used in this research and then present an evaluation of these techniques. Chapter 3 will describe the inadequacies of the ScriptEase Pattern Builder and the Code Viewer to detect potential conflicts and then introduce the Conflict Advisor with the Object-slot View to resolve this issue.

Chapter 2

Optimization

From a compiler's perspective, ScriptEase can be viewed as a code generator that transforms user-adapted patterns into scripts that run in video games. Since code is generated independently from different sources, it may have many redundancies as discussed in Section 1.7, leaving broad scope for optimization. This chapter will describe how to reduce lines of code by performing a post-processing transformation on the automatically-generated scripts in ScriptEase. The reason for focusing on code reduction rather than on code speed is discussed in Section 2.1.4.

2.1 Related Work

Compiler designers always wish their compilers were able to produce target code that is as good as manually-written code. However in reality, it is difficult to achieve this goal if a naive compiler is implemented. Therefore, a huge number of compiling techniques that transform the compiled code in order to improve its execution speed or reduce its space consumption, or both, have been invented. These code-improving transformations are called *optimizations* [1].

2.1.1 Criteria for Code-improving Transformations

Code-improving transformations (optimizations) should have the following properties described by Alfred V. Aho *et al.* [1].

First, an optimization "must preserve the meaning of programs". After an optimization is performed, the resulting program must produce exactly the same output as the original unoptimized program for any given input. This is known as a *safe* optimization. At all times compiler designers take the safe approach of missing an opportunity to optimize rather than risk modifying what the program is supposed to do.

Second, an optimization "must, on the average, speed up programs by a measurable amount". Not every optimization improves every program for every input. Occasionally an optimization may slow down a program for some of its inputs. An optimization is useful as long as on average it succeeds in improving the performance of many program on many inputs. Sometimes reducing

the space taken by the compiled code is more important to compiler designers that increasing the execution speed.

Third, an optimization "must be worth the effort". If a compiler designer expends the intellectual effort and makes the compiler expend the extra time optimizing programs but the effort is not repaid when the optimized program is executed, there is no reason to design and perform such optimizations.

Charles N. Fischer *et al.* further summarize the points into two criteria that can be used to judge an optimization: *safety* and *profitability* [13]. Optimizations guaranteed to always produce exactly the same results are *safe*. Optimizations are *profitable* if they can improve at least one aspect of a program's performance (e.g. speed, code size, cost billed to a user, system overhead).

An optimization can sometimes satisfy both criteria and multiple aspects of the profitability criterion. For example, it may safely eliminate an unnecessary instruction, which reduces program size and improves speed. However, some optimizations benefit us in one criterion or one aspect at the expense of another. Usually *safety* takes priority over all aspects of profitability. In this thesis, I always try to use *safe* optimizations and value reduction in code size more than other performance aspects as described in Section 2.1.4.

2.1.2 Peephole Optimization

Peephole optimization is a simple but effective optimizing technique that tries to improve the performance of the target program by identifying patterns (a short sequence of instructions) and replacing them by a shorter or faster sequence. This replacement may occur again on the new sequences if additional patterns are identified after one pass of this optimization. The term *peephole* represents a small, moving window on the target program, so peephole optimization is referred to as a local optimization technique. I present here some examples of typical peephole optimizations that I used in this research.

Constant folding Code for algebraic computations sometimes contains constant values that could be pre-computed at compile time. For example, the statement

```
a = 3 * 15 + 25;
```

can be replaced by the statement

```
a = 70;
```

Dead-branch elimination Another opportunity for peephole optimization is the removal of unreachable instructions. One common spot for this transformation is conditional branches like an `if` statement. For example, a large segment of code may be contained within an `if` branch for debugging purposes. The code may look like this in C:

```
1  #define debug 0
2  ...
3  if ( debug ) {
4    print debugging information
5  }
```

Since `debug` has a constant value of 0, the statements inside the `if`-branch are never going to execute. Thus all these statements are manifestly unreachable and should be eliminated.

Common subexpression elimination An expression E is called a *common subexpression* if E was computed previously and variables in E haven't changed their values since the previous computation [1]. Recomputing the expression can be avoided if we can use its previous value. In the following example:

```
a = 4 * i + 1;
b = 4 * i + a;
```

it can be transformed into the following code to improve efficiency:

```
tmp = 4 * i;
a = tmp + 1;
b = tmp + a;
```

2.1.3 Global Optimization

Peephole optimization is performed on basic blocks (i.e. peephole). A basic block is a code segment that has a single entry at the top and a single exit point at the bottom. There is another optimization that deals with the flow of control across basic blocks – global optimization. Peephole optimization is usually simple and straight-forward, while global optimization is difficult and potentially unsafe, requiring intensive preprocessing analysis. For example, optimizing the assignment of variables to registers across basic blocks proves rather difficult. There are many variables but few registers, so to minimize the overall overhead of loading/saving variables from/to registers, an enormous number of possible assignments must be considered. An example of a potentially unsafe transformation is *loop-invariant code motion*. If an expression is computed inside a loop and its value is invariant, it is more efficient to hoist this expression outside the loop and compute its value just once. However, this technique could be unsafe if the expression has side effects. For instance, if the evaluation of this expression causes program output, then removing it from the loop will change the output. Although being difficult and potentially unsafe, global optimization often offers greater benefits. Since it identifies redundancies on a broader scope compared to peephole optimization, more code can be positively affected. A selection of common global optimization techniques are presented here.

Function inlining For modern programmers, modularity is often considered a more important factor than efficiency. A collection of cleanly-defined functions is preferred over large main programs for improved readability and ease of maintenance. Unfortunately, function calls are expensive, involving scope change, call-site information storage, local variable allocation and argument transmission. To address this issue, function inlining (also known as inline expansion of function calls, or inline expansion for short) has been implemented in major compilers, so that programmers can still write code in a structured [12] way and rely on compilers to transform their code into unstructured but more efficient programs.

Inline expansion of a function call replaces a call to a function with the function body and handles variable scopes, parameters and return values. This transformation saves much of the overhead associated with function calls such as saving registers and creating new stack frames. Additionally, function inlining exposes the function body and parameter values, which makes other optimization techniques potentially applicable. For example, if the actual function arguments are constants, there could be opportunities for *constant folding* and *dead-branch elimination* to further optimize the inlined code.

Choosing the right candidate for function inlining requires careful consideration since inline expansion is mainly a space-for-time tradeoff – the expansion of a function call almost always takes more space than a call itself, except when a function is called only once, in which case both time and space are reduced. To prevent code explosion, usually only small functions (few lines of code) are considered good candidates for inline expansion. To maximize the efficiency gain, frequently called functions are the best candidates for the simple reason that the time saved will be leveraged manyfold. However, an extreme counter example would be recursive functions, where inline expansion may lead to disastrous results since every expansion potentially generates another expansion. In summary, frequently called non-recursive small functions or an once-called large functions are good candidates for function inlining.

Loop optimization Loops are often considered "hot spots" – a region of a computer program where most time is spent during its execution – for optimization since they indicate repetitious and time-consuming execution of instructions. A great number of optimization techniques have been developed to make the execution of loops faster, of which I present some here.

- loop interchange – by exchanging inner loops with outer loops, this technique can improve locality of reference if the outer loop involves indexing into an array.
- loop fusion – when two contiguous loops iterate the same number of times, their bodies can be combined to reduce loop overhead as long as they don't share each other's data.
- loop fission – this technique attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. This transformation can improve locality

of reference. For instance, when a loop iterates over two arrays, loop fission separates it into two loops, each referencing only one array. This is the opposite of loop fusion, so loop overhead should be compared to locality of reference to determine whether this technique is beneficial.

- loop unrolling – if the number of iterations is known at compile time, the loop can be expanded by duplicating the loop body multiple times and removing the loop head. This technique reduces the number of loop condition tests and makes the loop into a sequential set of instructions without jumps, which executes much faster on processors supporting pipelining. Unrolling increases code size to reduce execution time.
- loop-invariant code motion – if an expression is a constant inside a loop, this expression can be moved outside the loop so that it is evaluated only once. This could be unsafe if the expression has side effects as discussed previously, so careful analysis must be performed ahead of time.

Parallelization optimization With the advent of practical parallel processing, impressive performance gains have been achieved on multiprocessor machines [27]. Parallelization optimization refers to automatically converting sequential code into multi-threaded or vectorized code in order to utilize multiple processors, on which the program runs. This technique places most focus on loops, because loops are hot spots and the repetitive nature of loops can often be improved by parallelization. In order to perform this optimization, most compilers conduct two passes of analysis to determine if it is safe to parallelize the program and if it is worthwhile to parallelize it. There are numerous difficulties in parallelization optimization such as analysis of code dependence, indeterminable loop iterations and coordination of global resources.

Partial redundancy elimination (PRE) PRE is a compiler optimization that eliminates expressions redundant on some but not necessarily all paths through a program. It was first developed by E. Morel and C. Renvoise, which removes global common subexpressions and moves invariant computations out of loops at the same time [20]. Many PRE techniques based on this have been developed since then [4, 6, 15, 7].

2.1.4 Optimizing Scripting Languages

Since scripting languages are often interpreted from source code or byte-code[1], the process of compilation in traditional programming languages is absent and thus makes optimizing scripting languages different in at least two ways.

First, the efficiency criteria of optimization is slightly different, where code size – more specifically number of instruction lines – is a more important factor. The fact that scripts are interpreted

[1] Various forms of instruction sets designed for efficient execution by a software interpreter as well as being suitable for further compilation into machine code.

during execution means that fewer lines of code results in less overhead for the interpreter and will considerably improve the speed of the execution. Moreover, measuring the execution time of a script requires the support of the interpreter or the virtual machine on which the script is executed, which is not always available to designers of optimizers, so lines of code may often be a more important factor in determining how programs are improved by optimization.

Second, optimizing scripting languages has limited applicability. The techniques discussed previously take place before or during the translation of source code into native machine code and will take a certain amount of time. This time overhead is not much of an issue for traditional programming languages as a program only needs to be compiled once and can be run multiple times. But for scripting languages, if optimization is performed during interpretation, the time overhead may overcome the efficiency improvement gained from the optimization. Therefore, optimization for scripting languages is only practical in two situations: 1. when byte-codes are available for a scripting language, then optimization can take place before or after the source code is transformed into byte-codes; 2. when scripts are automatically generated, then optimization can be performed after or during the process of generation.

An example of the first situation is optimization for PHP [24] scripts. PHP is a widely-used general-purpose scripting language designed for producing dynamic web pages. Since it generally runs on a web server, there is a great demand for optimization. Since version 4 of PHP, a PHP parser produces byte-codes. Several commercial products have been developed such as Zend Optimizer [31] and PHP Accelerator [25]. PHP Accelerator's main feature is the employment of "compiled code caching" [17] to accelerate the execution of PHP byte-codes. In addition to "compiled code caching", it also performs a number of peephole optimizations on the compiled byte-codes like "elimination of unconditional jumps to other unconditional jumps or return statements, reordering of some code constructs to eliminate jumps, and removing unreachable code" [17]. PHP Accelerator claims to deliver at least a 2 to 4 times performance gain [17], and detailed benchmarks can be found at `http://www.php-accelerator.co.uk/performance.php`.

Google Web Toolkit (GWT) [14] is an example of the second situation – optimizing auto-generated scripts. GWT is a software tool that allows web developers to build and maintain JavaScript front-end applications in the Java programming language. Its compiling engine generates JavaScript from user-written Java code and optimizes the scripts to load and execute faster than equivalent handwritten JavaScript code. Common compiler optimizations can be seen in GWT such as dead-code elimination and function inlining. Unlike most optimizing compilers, GWT puts emphasis on the file size of scripts rather than lines of instructions, since data transmission is a crucial performance bottleneck in web applications, especially those that rely heavily on JavaScript. Therefore, GWT eliminates all unnecessary blank characters and renames most variables and functions using shorter names. GWT also has a feature called "deferred binding", where it generates many versions of JavaScript code for browser compatibility but loads only the appropriate version for a particular

client at runtime. By applying these techniques, GWT is able to produce compact code that can be quickly delivered over the Internet.

Biggar *et al.* systematically discuss the problem of compiling any scripting language to improve its performance [2]. They present an ahead-of-time compiler for PHP which compiles PHP scripts into C code, interfaces it with the PHP C API and then compiles it into an executable (or a shared library) by a C compiler. The link to the C API allows them to perform a number of transformations during compilation into C code, including constant-folding, pre-hashing, symbol-table removal and pass-by-reference optimization. They claim their approach is applicable for almost all scripting languages (PHP, Perl, Python, Ruby and Lua, excluding JavaScript).

2.2 Optimizing ScriptEase-generated NWScript

Optimizing ScriptEase-generated NWScript is applicable since it falls under the auto-code-generation situation, similar to GWT. With GWT, the goal is reducing the number of bytes of scripting code to a minimum. With NWScript, the goal is similar, reducing the number of NWN byte-codes compiled from the script. Since the byte-codes are essentially API calls, my optimizer tries to reduce the number of lines of NWScript code that will be compiled. Moreover, safety is always maintained in my optimizations.

Many traditional compiler optimization techniques are used in this research: constant folding, dead-branch elimination and function inlining. However, due to the characteristics of scripting code in video games, many optimization techniques do not apply. First, the input code is a scripting language that is interpreted by the game engine. Therefore low-level issues that involve memory allocation and instruction scheduling are very difficult to tackle, given that I do not have direct access to the game engine. Second, since the scripts required to control the story in video games rarely requires the use of loops, the tremendous amount of loop optimizations in compilers are inapplicable to my problem. Third, *NWN* and NWScript – the game and the scripting language with which I am working – do not support parallelization, which renders related techniques irrelevant.

On the other hand, I found places in ScriptEase which require optimizations uncommonly seen in standard compiler optimization. The most significant one is eliminating redundant code by exploiting the fact that ScriptEase generates redundant code from different patterns and inserts it into one single script. The next section will explain in detail the optimization techniques used in this research.

2.3 Optimization Techniques Used

2.3.1 Constant Folding

This is a common compiler optimization technique.

```
if ( ! TRUE )                    if ( FALSE )
    return;                          return;
```

Figure 2.1: Example of constant folding

In Figure 2.1, the expression `!TRUE` on the left side of the figure can be evaluated at compile time and therefore should be replaced with its value `FALSE`, as shown on the right side.

2.3.2 Dead-branch Elimination

This is a common compiler optimization technique.

```
1 if( FALSE ) return;
```

Code 2.1: Example of dead-branch elimination

In Code 2.1, the condition for this if-statement is the constant value `FALSE`, so the subsequent branch will never be executed. Thus the entire statement can be removed without affecting the correctness of the program.

In general, this transformation can be performed if we have code in the form shown in Code 2.2.

```
1 if ($ConstantExpression$) {
2     // Do something
3 } else {
4     // Do something else
5 }
```

Code 2.2: General form of dead-branch elimination

If `$ConstantExpression$` is evaluated to be true we can eliminate the else-branch and leave only the *`//Do something`* part. Otherwise, only the *`//Do something else`* branch is kept.

As shown in Code 2.3.1 and Code 2.1, this transformation can be done jointly with constant pre-computation: whenever there is an if-statement, a pre-computation is performed if necessary and then the if-condition is checked to see whether there is any dead branch to remove.

2.3.3 Function Inlining

This is a common compiler optimization technique.

```
1 void SE_Ac_OpenCloseDoorTodo(object param_1,
2     object param_2, int param_3);
3 void SE_Ac_OpenCloseDoorTodo(object param_1,
4     object param_2, int param_3) {
5     if (param_3)
6         AssignCommand(param_1, ActionOpenDoor(param_2));
7     else
8         AssignCommand(param_1, ActionCloseDoor(param_2));
9 }
```

```
10 void Useplaceable_0() {
11     /*---Code omitted---*/
12     // Stone Door* Open or Closes Stone Door
13     SE_Ac_OpenCloseDoorTodo(StoneDoor_SE2, StoneDoor_SE2,
14         OpenorClose_SE5);
15 }
16 void main() {
17     Useplaceable_0();
18 }
```

Code 2.3: Example of function inlining

In traditional compiler optimization, the criteria of selecting candidates for function inlining are 1. they are executed many times, e.g. in a loop, in order to achieve maximum performance gain and 2. they are usually short, in order to prevent code explosions. But in ScriptEase, loops are rare and most functions defined in one script are called only once, so code explosion is usually not an issue. The exception is that ScriptEase uses some `#include` files, where functions may be used in many scripts, therefore these functions are not inlined. Thus I decided to inline as many functions defined in the script as possible (see exceptions below). With the removal of function declarations, function definitions and function calls, we are guaranteed to have fewer lines of code; and less resources will be consumed at runtime posterior to this transformation since fewer lines of code and fewer function calls will be interpreted.

In Code 2.3, `main()` calls `Useplaceable_0()`, and then `Useplaceable_0()` calls `SE_Ac_OpenCloseDoorTodo()`. Both functions are defined in this script. According to our notion of function inlining, both `Useplaceable_0()` and `SE_Ac_OpenCloseDoorTodo()` are going to be inlined. The optimized code is shown in Code 2.4, where both `Useplaceable_0()` and `SE_Ac_OpenCloseDoorTodo()` are removed and their function bodies are inserted into `main()`. The 18 lines in Code 2.3 are reduced to the 8 lines in Code 2.4, and 2 fewer function calls are made.

```
1  void main() {
2      /*---Code omitted---*/
3      // Stone Door* Open or Closes Stone Door
4      if (OpenorClose_SE5)
5          AssignCommand(StoneDoor_SE2,
6              ActionOpenDoor(StoneDoor_SE2));
7      else
8          AssignCommand(StoneDoor_SE2,
9              ActionCloseDoor(StoneDoor_SE2));
10 }
```

Code 2.4: Example of optimized code after function inlining

The general form of this transformation is illustrated in Code 2.5, with the optimized result shown in Code 2.6. The example in Code 2.3 and Code 2.4 illustrates two applications of this general form in Code 2.5.

```
1  void function(Type param1, ...);
2  void function(Type param1, ...) {
3      // implementation of function
4  }
5  void main() {
6      function(arg1, ...);
7  }
```

Code 2.5: General form of function inlining

```
1  void main() {
2      // implementation of function
3      // param1, ... are replaced by arg1, ...
4  }
```

Code 2.6: Result of function inlining

As stated above, the goal is to inline all functions defined in the script, but three exceptions apply.

1. The entry point of a script is not inlined. *main()* is the entry point of a regular script in NWScript and *StartingConditional* is used in a conversational filter script. These two functions cannot be inlined since they are invoked by the game engine.

2. Functions with *early returns* are not inlined, since my technique for function inlining does not support early returns: the way functions with a return value are inlined in my implementation is to declare a variable of the return type and then replace all *return* statements in the function body by assignment statements that bind the returned values to the declared variable. Later when this function is called as an expression, a reference to the declared variable replaces the function call. An example is shown in Code 2.7 and the optimized code is shown in Code 2.8.

```
1  int function(int a) {
2      return a+1;
3  }
4
5  void main() {
6     SomeAPI(function(9));
7  }
```

Code 2.7: Example of a function with a return value

```
1  void main() {
2      int function_0;
3      function_0 = 9+1;
4
5      SomeAPI(function_0);
6  }
```

Code 2.8: Result of inlining a function with a return value

In Code 2.7, `function()` is declared and later called in `main()`. After inlining the function, in Code 2.8, `function()` is removed and its function body is placed inside `main()`. Since `function()` has `int` return type, an integer `function_0` is declared right before the original function body. Then, all return statements in the body are replaced with assignments to `function_0`. And finally, the call to `function()` in `main()` is now a reference to the variable `function_0`, which represents the original returned value of the inlined function. Further optimization could be performed as well, for example, constant folding can be applied to `function_0 = 9+1;`.

This algorithm works fine in most cases. However, when the inlined function has early returns, this transformation is unsafe – the optimized code may not produce the same output as the original program. An early return is a `return` statements inside a branch (e.g. if-statement, loop) of a function body and it exits the function early so that the statements inside the function following the `return` statement will not be executed. In Code 2.9, the statement `return 1` inside the if-statement is an early return. When this script runs, if the condition `a > 0` is true, the function will end without executing any statement following `return 1` and thus `return 0` will not be executed. If the code is optimized using the inlining algorithm proposed previously, the resulting program's output will be different from that of the source script. As shown in Code 2.10, `return 1` is replaced by `function_0 = 1` and the latter does not skip the execution of the next statement: `function_0 = 0`. In fact, the semantics of the optimized code is that *no matter what the value of a is, function_0 is always going to be 0*, which is different from the semantics of the original code – *return 1 if a is greater than 0, return 0 otherwise*.

```
1  int function(int a) {
2      if ( a > 0) {
3          return 1;
4      }
5      return 0;
6  }
7
8  void main() {
9     SomeAPI(function(9));
10 }
```

Code 2.9: Example of a function with an early return

```
1  void main() {
2      int function_0;
3      if ( 9 > 0) {
4          function_0 = 1;
5      }
6      function_0 = 0;
7
8      SomeAPI(function_0);
```

```
9 }
```

Code 2.10: Result of inlining a function with an early return

There are three ways to solve the issue of early returns in function inlining. First, a `goto` statement can be used to skip the statements following the early return. In a normal compiler, which generates machine language code, using `goto` is fine. However, we are performing script-to-script translation in a high-level scripting language, undisciplined use of `goto` complicates the structure of code so its usage is not recommended [11]. Whether `goto`s are harmful or not is irrelevant since this command is not supported in NWScript. Second, a code block containing early returns can be transformed so that the skipped code is moved into a branch which is mutually exclusive to the branches where the early returns take place. For example, Code 2.11 shows how Code 2.7 can be transformed by adding an `else` clause. With this transformation, all early returns are eliminated and the inlining algorithm will be a safe transformation.

```
1  int function(int a) {
2      int tmp;
3      if ( a > 0) {
4          tmp = 1;
5      } else {
6          tmp = 0;
7      }
8      return tmp;
9  }
10
11 void main() {
12    SomeAPI(function(9));
13 }
```

Code 2.11: Transformed code with early return issue resolved

However, the introduction of early-return-eliminating transformations does have a critical side effect: an increase in code size, especially when early returns are found in loops or multiple positions within one single function. Although loops are rare in NWScript, they can occur. Since code size is what my optimizations are trying to reduce, this solution is not viable.

The third possible solution is naive but effective: do not inline functions with early returns. By doing this we sacrifice some efficiency gains for the safety criterion, which is the trade-off I decided to make.

3. Functions called by **Command* are not inlined. Usually in a programming language, a function with `void` return type cannot be invoked as an expression since it does not return anything. But in NWScript, there is a special type of function called *action*, which doesn't return a value (so it is declared as a `void` function) but can still be employed as an expression by

Command APIs (namely *AssignCommand*, *DelayCommand* and *ActionDoCommand*). For example, in Code 2.12 `someAction()` is an *action* function and used as an expression in an *AssignCommand* call. In NWN, *Command functions are used when an action that takes a long time should be performed by an object when the other action being performed by that object are finished. If an action occurs directly in a script rather in a *Command call, it is executed to completion before the script can continue. In essence, each object maintains a queue of actions that are performed in order.

```
1  void someAction() {
2      /* Omitted code */
3      ActionRest();  // Action API
4  }
5
6  void main() {
7      AssignCommand(GetNearestObjectByTag("human"),
8          someAction());
9  }
```

Code 2.12: Example of "action" functions

Since there is no value returned in the invocation to an action function, inlining this function is difficult, if not infeasible. If I were to inline such a function, I would have to extract the action API in that function (such as `ActionRest()` in Code 2.12), replace the function call (`someAction()`) with this action API call and move the rest of the function body (`/* Omitted code */`) before `AssignCommand()`. This assumes that none of the omitted code is an action. If so, that action would have to be placed in its own `AssignCommand` call. This process is further complicated if the omitted code in Code 2.12 has side effects on any argument of `AssignCommand()` preceding `someAction()` such as `GetNearestObjectByTag()`. For these reasons, I decide not to inline action functions called by **Command* APIs.

2.3.4 Redundant Code Elimination

Redundant code elimination is related to common subexpression elimination. It applies when the same statement appears twice due to function inlining. Although related to PRE, my technique does not require data-flow analysis, which is needed in most PRE techniques.

Since ScriptEase generates code snippets from independent patterns and combines them into one single script if they are attached to the same event slot of a unique object, redundant code can be generated from these independent sources. An example of such redundancy was presented in Section 1.7. In Code 1.2, the redundant lines of code are highlighted in bold. Redundant code elimination combines those identical expressions and statements.

Redundant code elimination takes place after function inlining: in Code 1.2, the two functions `Useplaceable_0()` and `Useplaceable_1()` are going to be inlined first so that their code

bodies are both placed in the scope of main. Then elimination is performed to remove potential redundant variable declarations, assignments and API calls.

- Variable declarations

  If a variable with the same name and type is declared before, I will eliminate this declaration statement because the previous variable can be re-used. Since functions are inlined strictly in the order they are called, when a new variable declaration appears in the code, it must be originally from the body of a different function. Therefore, it is safe to assign new values to the variable and reference it afterwards. The reason I chose variables with the same name to optimize is because ScriptEase has a pattern of naming variables when generating code, where identical variable names indicate a similar task. For instance, User_SE0 is almost always used to represent the user of a placeable. This convention helps me perform the next transformation - redundant-variable-assignment elimination.

- Variable assignments

  When the left-hand side of an assignment is a variable that appears in the code before, it is likely that there are redundant variable assignments. In Code 1.2, both functions Useplaceable_0() and Useplaceable_1() have an assignment statement ThePlaceable_SE1 = OBJECT_SELF. When the two functions are merged, only the first copy of this statement should be kept as the second one is merely a futile repetition. Since redundant variable declarations are eliminated, later references to ThePlaceable_SE1 all refer to the first value assigned. When deciding whether an assignment is redundant I compare the right-hand side of the statements and if necessary this comparison will be performed recursively. Recursion is required when the right-hand side is an expression containing other variables, which are then compared to their previous values. An example is shown in Code 2.13.

```
1  void main() {
2      int a, b, c;
3      a = 1;       // 1st assignment of a
4      b = 2;       // 1st assignment of b
5      a = 1;       // 2nd assignment of a
6      b = 3;       // 2nd assignment of b
7      c = a + b;   // 1st assignment of c
8      c = a + b;   // 2nd assignment of c
9      a = 2;       // 3rd assignment of a
10     c = a + b;   // 3rd assignment of c
11 }
```

Code 2.13: Example of redundant-variable-assignment elimination

  In Code 2.13, three integers are declared in the beginning: a, b and c. Some statements involving references to these three variables are omitted for clarity. For example, all references to b between Line 4 and Line 5 are omitted. Line 3 assigns the value 1 to a and Line 4 assigns

2 to `b`. Line 5 is a redundant assignment because its right-hand side value is the same as the previous assignment of `a` in Line 3, while Line 6 is not redundant because its right-hand side value is different from that of Line 4 (recall that references to `b` between Line 4 and Line 5 are not displayed). Line 7 assigns `a + b` to `c`. Line 8 is therefore redundant because the value of neither `a` or `b` is changed between Line 7 and Line 8. Line 9 is not redundant because the value of `a` changes from 1 to 2. Line 10 is not redundant either due to the change of `a`'s value in Line 9. The optimized code is shown in Code 2.14.

```
1  void main() {
2      int a, b, c;
3      a = 1;
4      b = 2;
5      // a = 1;        //Redundant
6      b = 3;
7      c = a + b;
8      // c = a + b;    //Redundant
9      a = 2;
10     c = a + b;
11 }
```

Code 2.14: Result of redundant-variable-assignment elimination

- API calls

  In NWScript, there are a large number of API functions that are implemented in the game engine and provided for script writers to use. They perform various tasks from accessing attributes of a game object to controlling in-game actions. There is an unofficial website called NWNLexicon [21] that provides a general purpose reference to the NWScript API.

  The transformation for eliminating redundant API calls is similar to that of variable assignments because I focus on API calls that appear on the right-hand side of an assignment. In Code 1.2, `User_SE0 = GetLastUsedBy()` appears in both functions and therefore the one that occurs later is redundant. The process of determining redundancy is similar to that of variable assignment and recursion is used when the API function takes non-constant arguments.

  However, one more factor needs to be taken into consideration when performing this transformation. Among all the API functions that are accepted on the right-hand side of an assignment statement (i.e. functions with a return value), some functions may return different values given the same arguments. An example would be the random number generating function, `random (int max)`, which generates a random number betweeen 0 and `max`-1. The output of this function is different every time it is called, and therefore it cannot be optimized if one is to ensure that the program behaves exactly the same as it does before transformation. We name this type of API function a *non-deterministic* function as it may produce different results when

called with invariant arguments multiple times (e.g. `random(10)`). The rest of the API functions are called *deterministic* functions. For instance, the function `GetLastUsedBy()` always returns the last user of a placeable within the scope of an *OnUsed* event, which is consistent throughout one execution instance. Moreover, a script runs to conclusion without interruption, so from one point in a script to another point, state cannot change except in non-deterministic API calls. The solution is to maintain a list of *non-deterministic* API functions and exclude them from API-call elimination.

The optimized version of Code 1.2 is shown in Code 2.15 with all optimization techniques applied. Lines in bold indicate where tranformations take place.

```
1  #include "i_se_aux"
2  //@Optimization performed:  function declaration removed
3  /*void SE_Ac_OpenCloseDoorTodo(object param_1,
4      object param_2, int param_3);*/
5
6  /*void SE_Ac_ShowImpactVisualEffectOnCreature(int param_1,
7      object param_2);*/
8
9  //object SE_Ac_SpawnCreatureNearObject(string param_1, object param_2);
10
11 //void Useplaceable_0();
12
13 //void Useplaceable_2();
14
15 void main() {
16
17     // The following are all of the variables used in this situation
18     int OpenorClose_SE5;
19     object ThePlaceable_SE1;
20     int UnlockorLock_SE4;
21     object User_SE0;
22     object StoneDoor_SE2;
23     int DoorLocked_SE3;
24
25     // When Stone Door Lever is used
26     //@Optimization performed:  dead branch removed
27     //@Optimization performed:  constant pre-computation
28     /* if( FALSE ) return;*/
29
30     // Get the object with tag "UA_StoneDoor"
31     StoneDoor_SE2 = GetNearestObjectByTag("UA_StoneDoor");
32     if (StoneDoor_SE2 == OBJECT_INVALID )
33         StoneDoor_SE2 = GetObjectByTag("UA_StoneDoor");
34
35     // Define Door Locked as whether Stone Door is locked
36     DoorLocked_SE3 = GetLocked(StoneDoor_SE2);
37
38     // Define Unlock or Lock as the opposite of Door Locked
39     UnlockorLock_SE4 = ! DoorLocked_SE3;
40
```

```
41      // Define Open or Close as the opposite of Unlock or Lock
42      OpenorClose_SE5 = ! UnlockorLock_SE4;
43
44      // Define User as the user of Stone Door Lever
45      User_SE0 = GetLastUsedBy();
46
47      // Define The Placeable as the object the event is fired on
48      ThePlaceable_SE1 = OBJECT_SELF;
49
50      // Main code body - checks conditions and executes actions
51
52      // Animate The Placeable* to Activate
53      AssignCommand(ThePlaceable_SE1,
54          ActionPlayAnimation(ANIMATION_PLACEABLE_ACTIVATE));
55
56      // Animate The Placeable* to Deactivate
57      AssignCommand(ThePlaceable_SE1,
58          ActionPlayAnimation(ANIMATION_PLACEABLE_DEACTIVATE));
59
60      // Unlock or Lock Stone Door
61      SetLocked(StoneDoor_SE2, UnlockorLock_SE4);
62
63      // Stone Door* Open or Closes Stone Door
64      //@Optimization performed:  function inlining
65      /*SE_Ac_OpenCloseDoorTodo(StoneDoor_SE2, StoneDoor_SE2,
66          OpenorClose_SE5);*/
67      if (OpenorClose_SE5)
68          AssignCommand(StoneDoor_SE2, ActionOpenDoor(StoneDoor_SE2));
69      else
70          AssignCommand(StoneDoor_SE2, ActionCloseDoor(StoneDoor_SE2));
71
72      // The following are all of the variables used in this situation
73      object SpawnedCreature_SE2;
74
75      //@Optimization performed:  redundant variable declaration removed
76      /*object ThePlaceable_SE1;*/
77
78      //@Optimization performed:  redundant variable declaration removed
79      /*object User_SE0;*/
80
81      // When Stone Door Lever is used
82      //@Optimization performed:  dead branch removed
83      //@Optimization performed:  constant pre-computation
84      /* if( FALSE ) return;*/
85
86      // Define User as the user of Stone Door Lever
87      //@Optimization performed:  redundant assignment removed
88      /* User_SE0 = GetLastUsedBy();*/
89
90      // Define The Placeable as the object the event is fired on
91      //@Optimization performed:  redundant assignment removed.
92      /* ThePlaceable_SE1 = OBJECT_SELF;*/
93
94      // Spawn Spawned Creature from Evil Penguin near The Placeable
```

```
95      //@Optimization performed:  function inlining
96      /*SpawnedCreature_SE2 = SE_Ac_SpawnCreatureNearObject(
97          "penguin002", ThePlaceable_SE1);*/
98      object SE_Ac_SpawnCreatureNearObject_0;
99      location loc = GetLocation(ThePlaceable_SE1);
100     SE_Ac_SpawnCreatureNearObject_0 =
101         CreateObject(OBJECT_TYPE_CREATURE, "penguin002", loc);
102     SpawnedCreature_SE2 = SE_Ac_SpawnCreatureNearObject_0;
103
104     // Show the Unsummon impact visual effect on Spawned Creature
105     //@Optimization performed:  function inlining
106     /*SE_Ac_ShowImpactVisualEffectOnCreature(VFX_IMP_UNSUMMON,
107         SpawnedCreature_SE2);*/
108     effect veffect = EffectVisualEffect(VFX_IMP_UNSUMMON);
109     ApplyEffectAtLocation(DURATION_TYPE_INSTANT, veffect,
110         GetLocation(SpawnedCreature_SE2));
111 }
```

Code 2.15: Optimized code

2.4 Script Transformation Implementation

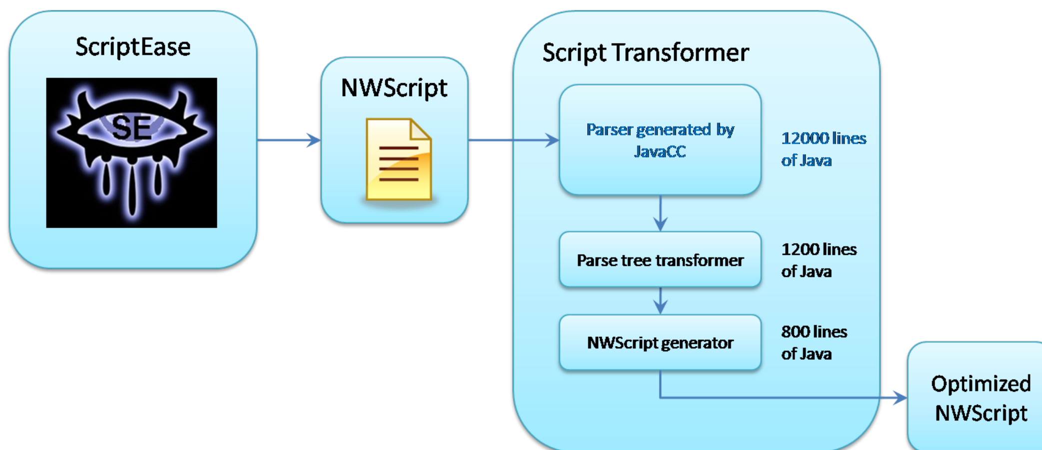

Figure 2.2: Architecture of the code transformer

Figure 2.2 illustrates how code transformation is implemented in ScriptEase. First, ScriptEase generates NWScript from patterns created by game designers. Then the scripts are processed by my script transformer, implemented in Java. The script transformer consists of three parts: an NWScript parser – generated by JavaCC (which is shown in dark blue in the diagram), a parse tree transformer and an NWScript generator – both implemented by me. Finally, optimized NWScript is generated and attached to the corresponding event-slots. The total Java code I've written is approximately 2000 lines.

2.5 Evaluation

Direct measurement of NWScript execution time is not available since no execution time API calls are provided by the game engine. Since measuring FPS (frame per second) of game runs could be misleading as most execution time is spent on graphics, I decided to measure code reduction to evaluate the optimization. Code transformation was performed on a number of modules scripted by ScriptEase, listed in Table 2.1. The first column lists the names of the modules used. The second column presents the number of ScriptEase pattern instances found in the module, and the third column displays the number of scripts generated from these patterns.

Module name	Number of pattern instances	Number of Scripts
The Summer Story	400	360
The Summer Story – Phase 1	10	8
Castle	8	21
The Room	2	1
Quest_Test2 Complete	3	7

Table 2.1: Modules used to evaluate the transformations

The Summer Story is the largest module used in this evaluation in terms of file size, number of patterns and number of scripts. It was originally made to demonstrate ScriptEase's ability to craft an entire story with patterns. The version of this module used here contains 400 pattern instances (including *encounter* and *quest* patterns described in Section 1.5) and 360 scripts are generated by ScriptEase.

The Summer Story – Phase 1 is a subset of the previous module and features the first phase of the story. There are 10 patterns and 8 scripts in the module. 21 scripts are generated from 8 patterns. All of the scripting code for this module and the optimized scripting code are provided in Appendix A.

Castle is a module containing pattern instances from three categories – *encounter*, *quest* and *behaviour* patterns. This module is used by the ScriptEase team to demonstrate the variety of patterns that can be used in one game story.

The Room is a tiny module with only 2 encounter patterns and 1 script. This is an ideal scenario for optimization as both encounters are attached to the same event-slot and generate code into the one single script.

Quest_Test2 Complete is a module made by Marcus Trenton [29] to test quest patterns in ScriptEase. This module has 3 quest pattern instances and 7 generated scripts.

To ensure the safety of code transformations, I tested all modules by playing through them in *NWN*. During testing, I discovered bugs caused by unsafe code transformations in my methods and corrected them. Inlining functions with early returns is an example of an unsafe operation that was detected during testing. Beside empirical testing, there is no formal proof of the safety of my code transformations.

Code reduction rate is measured on both the NWScript code (NSS file) and the compiled byte-codes (NCS file). For NSS files, lines of all scripts in one module were summed before and after optimization. Only non-comment and non-blank lines of code were counted. Then code reduction rates were calculated. For NCS files, bytes of compiled byte-codes were summed instead. Only bytes compiled from the generated scripts were counted and those from `#include` files were subtracted. The result is shown in Table 2.2.

Module name	Total lines of NSS			Total bytes of NCS		
	Before optimization	After optimization	Reduction rate	Before optimization	After optimization	Reduction rate
The Summer Story	11485	8547	25.6%	226813	181561	20.0%
The Summer Story – Phase 1	271	186	31.4%	4244	3678	13.3%
Castle	2032	1758	13.5%	68026	70260	-3.3%
The Room	65	36	44.6%	1155	776	32.8%
Quest_Test2 Complete	662	341	48.5%	16561	15958	3.6%
Total	14244	10682	25.0%	312555	268555	14.1%

Table 2.2: Evaluation of code transformation on ScriptEase-scripted NWN Modules

The second and third columns indicate the total number of lines (non-blank, non-comment) in the NSS files before and after optimization. The fourth column gives the calculated code reduction rate of the NSS files, which is *(lines before optimization - lines after optimization) / lines before optimization*. Similarly, the last three columns show the measurements for the NCS files.

The Summer Story is a representative test case due to the enormousness of the module. After applying the optimization, I have reduced the lines of script by 25.6% and byte-code size by 20.0%.

As mentioned above, the optimization should achieve the best performance on *The Room* and it did. For this module, the optimizer has removed 44.6% redundancy in scripts and truncated 32.8% byte-codes.

Castle is an extreme case. The raw scripts have been optimized by only 13.5%, and the compiled code has actually grown after optimization (negative reduction rate). The relatively poor performance in optimizing NSS files could be explained by the presence of *behaviour* patterns in this module. The reliance of *behaviours* [9] on action queues which make frequent use of **Command* (`AssignCommand`, etc.) API calls reduces many opportunities for function inlining as decribed in Section 2.3.3 on Page 26. In addition to that, *behaviours* usually consist of multiple phases, resulting in `if`-branches surrounding function calls in the implemented code, which makes *redundant code elimination* discussed in Section 2.3.4 hardly applicable. These traits of behaviour pattern also account for the deteriorating result of NCS files as only the byte-codes related to behaviours have grown in size after being optimized.

Quest_Test2 Complete is another interesting case. The reduction of scripts is huge – 48.5%, while

the byte-code reduction tells us the optimization result is not that significant – only 3.6%. Unlike behaviours in *Castle*, the compiled files in this module all shrink in size. However, the reduction is little compared to the huge size of the original byte-codes.

By summing all the lines of code and number of bytes over the four modules (*The Summer Story – Phase 1* is not summed as it is already included within *The Summer Story*), before and after optimization, we obtain an overall code reduction rate of 25.0% on generated scripts and 14.1% on compiled byte-codes. This result supports my thesis that optimization can be used to significantly reduce the size of NWScript generated by ScriptEase.

2.6 Summary

This chapter first introduced related work in compiler optimization and specifically scripting language optimization. Then I analyzed the applicability of existent compiler optimization techniques and the challenges of optimizing NWScript. The majority of this chapter described in detail the optimization techniques used in this research, including constant folding, dead-branch elimination, function inlining and redundant code reduction. Finally, an evaluation of the optimization was given based on the code reduction rate on ScriptEase-scripted *NWN* modules, and an overall performance of 25% redundant script removal and 14% byte-code compression was achieved.

The next chapter will discuss another contribution of this thesis – the Conflict Advisor.

Chapter 3

Conflict Advisor

The previous chapter attempted to address the issue of potential inefficiencies in ScriptEase-generated code by means of performing optimizations. However, there is another peril of generating event-based scripts from independent sources – potential conflicts, as discussed in Section 1.4. A specific example of conflict in ScriptEase was given in Section 1.8, where an encounter needed to computer the door nearest to the lever that had been destroyed by an other encounter on the same lever. The second contribution of this thesis is introducing an alternative view to the current ScriptEase default interface. This new view is designed to assist game designers in detecting potential conflicts.

3.1 Related Work

3.1.1 ScriptEase Pattern Builder

Since ScriptEase is a software designed for non-programmers to design game stories, a user-friendly graphical user interface (GUI) is required to enhance usability.

Figure 3.1 illustrates the main GUI of ScriptEase, entitled *ScriptEase Pattern Builder*. When a user (a game designer) authors a module (create, modify or delete patterns in the module), this is where most user operations take place. The main panel (below the menu bar) displays all pattern instances in the opened module as a tree, where each pattern instance (hence we'll call it pattern for short) is represented as a node. These nodes can be expanded to show components inside a pattern, e.g. conditions and actions, introduced in Section 1.6. In a large module – with more than 100 patterns – it is recommended to organize patterns into folders, which are also nodes in the tree, just like the folder-file hierarchical structure commonly used to represent file systems. Figure 3.2 shows the folder structure in the module *The Summer Story*, which contains 400 patterns. This module is organized into folders such as *Phase 1*, *Phase 2*, etc., and these folders may contain sub-folders such as *Troll* etc.

Nodes in this tree view are not ordered and a user can freely move any node to anywhere on the same level (cross-level movement is also possible but is not germane to this discussion) without changing the story-line. However, the order of nodes *does* affect the line order of the scripting code

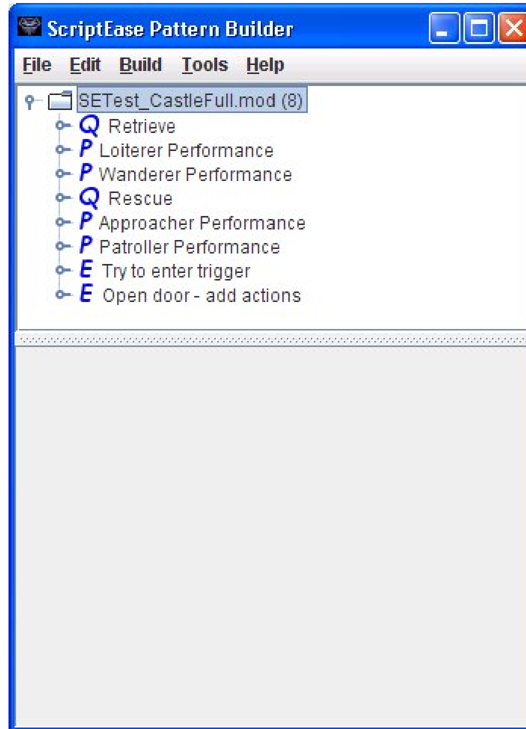

Figure 3.1: ScriptEase Pattern Builder

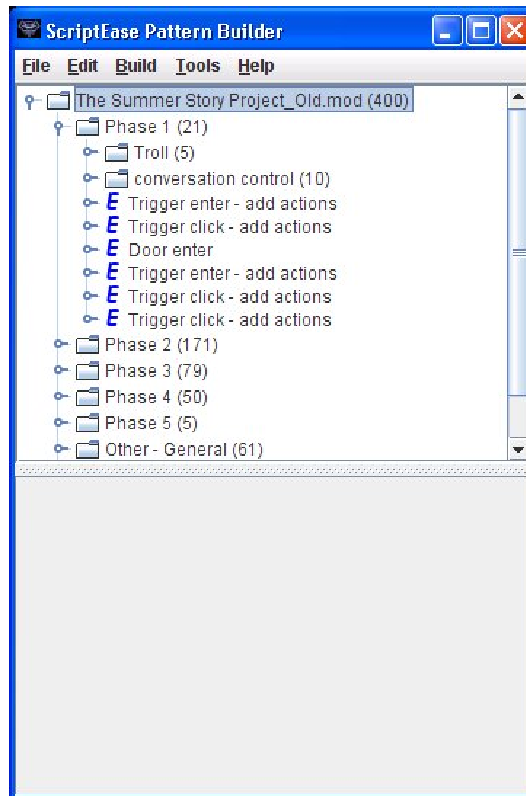

Figure 3.2: Folders in ScriptEase pattern builder

generated to represent them. However, users are unaware that the order of nodes matters. Thus no assumption should be made about the order of code generation when authoring a module. In other words, the user is supposed to treat all patterns as parallel to each other.

This is not an issue when the patterns are attached to different event-slots, which is usually the case. However, in the presence of some situations such as those described in Section 1.4, potential conflicts could arise and hinder the authoring process. The Pattern Builder offers no help in detecting such conflicts because it simply lists all patterns in parallel without providing any hint of interrelationships among them. It is true that the user can expand each pattern in the module to examine its even-slot in order to locate where potential conflicts could happen. However, this tedious process is undesirable when the module grows to contain hundreds of patterns, since the possibility of missing the target during manual inspection would be high.

Although the default pattern tree view is not helpful in detecting potential conflicts in ScriptEase, it still earns its place as a view that allows users to freely organize their patterns into folders, offering an interface similar to the file management GUI in major operating systems. Therefore, an alternative view would be beneficial if it is able to assist users to detect potential conflicts, as a supplement to the frequently-used default view.

3.1.2 ScriptEase Code Viewer

As revealed in Section 1.4, the root of potential conflicts in event-based video games is code generated from independent sources that interacts with each other. One way of locating potential conflicts would be to present the scripts generated from a pattern in the interface. Unfortunately, to make effective use of this technique, the user is required to read and debug the scripting language. This functionality was actually implemented in ScriptEase already, as shown in Figure 3.3 and Figure 3.4.

Figure 3.3 demonstrates how to view the code associated with an encounter pattern from the right-click menu. A new window will appear, shown in Figure 3.4, entitled *Code Viewer*. Complete code generated from the event-slot that this encounter is attached to can be found in the Code Viewer window. This means that not only code generated from this specific encounter, but all code found in the same script attached to the same event-slot (e.g. *OnOpen* on a door object) is displayed here. The Code Viewer is convenient since the entire script on a specific event-slot is presented so that all potentially conflicting spots can be traced. With the Code Viewer, designers are no longer required to look through the lengthy list of all patterns in a module. The Code Viewer would be useful if a designer already knew which pattern was involved in a conflict and, of course, if the designer was able to understand the scripting language.

However, the design goal of ScriptEase is to enable game authors with no programming background to write game stories. Thus, we cannot assume designers are able to understand the scripting language presented in the Code Viewer, specifically NWScript in this case. The Code Viewer is intended for designers who know NWScript and programmers who wish to make manual changes

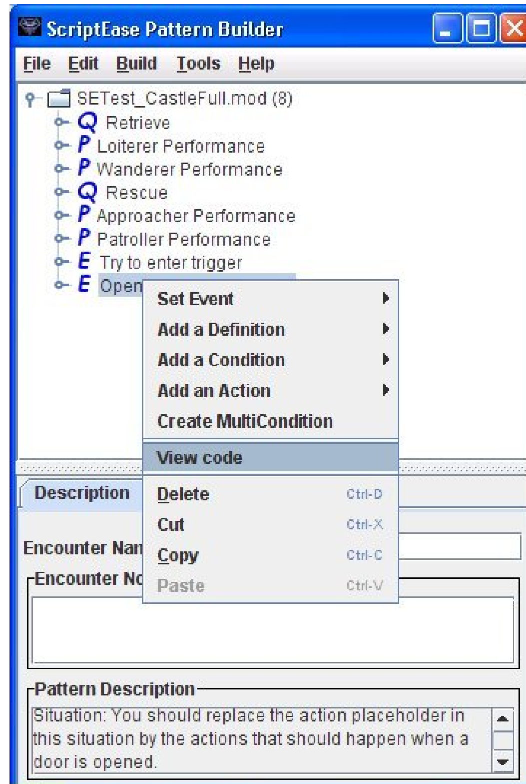

Figure 3.3: "View Code" in ScriptEase

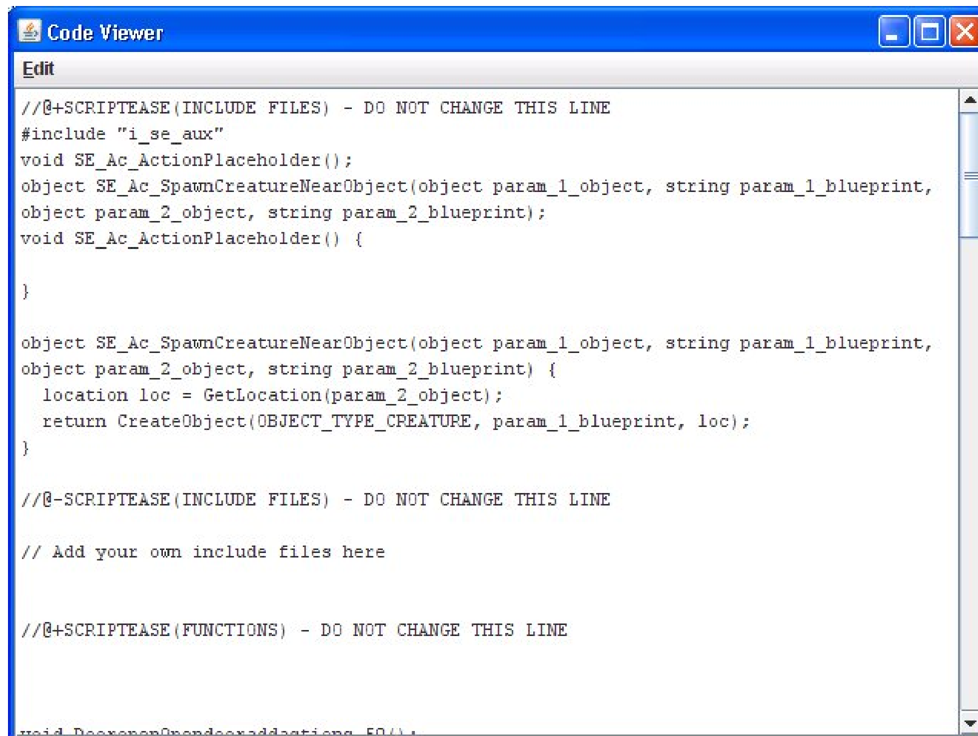

Figure 3.4: ScriptEase Code Viewer

to the generated scripts. They can view the code without leaving ScriptEase. The Code Viewer is also used by ScriptEase implementers to debug ScriptEase. Therefore, a more intuitive interface is needed for general users of ScriptEase to identify potential conflicts. Even if a designer has programming experience, viewing code is pattern-specific and the designer would have to know which pattern contains conflicts to use the Code Viewer. In other words, if it is only known that there are potential conflicts somewhere in a module, locating them could still be very difficult with the Code Viewer.

3.2 Conflict Advisor with Object-slot View

Having studied the inadequacies of both ScriptEase Pattern Builder and the Code Viewer in detecting potential conflicts, I identified three characteristics my proposed *Conflict Advisor* should have:

1. A graphical interface, as opposed to the Code Viewer's plain form of scripts.

2. The ability to locate potentially-conflicting patterns, as opposed to the Pattern Builder's natural or user-determined organization.

3. The potential to suggest conflicts, as opposed to the Code Viewer's reliance on designer knowledge of which patterns cause conflicts.

To fulfill the first requirement, I decided to keep the form of the tree view found in the Pattern Builder, so that a familiar GUI was retained to ease the process of learning. Then the question became how to present the pattern collection in a module, still in the form of a tree, to provide assistance in locating potential conflicts.

3.2.1 A Tree Representation

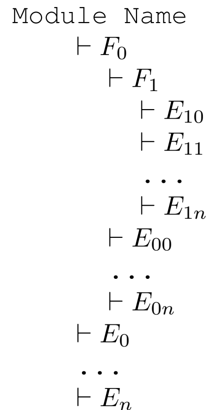

Figure 3.5: Tree view in the Pattern Builder

For clarity and simplicity of discussion, I have generalized the tree view in the Pattern Builder and as presented in Figure 3.5. "F" represents a folder and "E" represents a pattern (e.g. encounter pattern). The first subscript of a pattern indicates the folder that contains it and the second subscript represents its 0-based position in the folder. For example, E_{11} is in Folder F_1 and is in Position 1 (after Position 0) in its folder. This abstract tree illustrates the fact that in the Pattern Builder, the user has full control over where the patterns and folders are located and how many of them can be contained in any arbitrary folder.

3.2.2 The Object-slot View

To answer the question of how to help detect potential conflicts, we must revisit the event-based model in video games, as discussed in Section 1.2. In this model, each game object has a pre-defined list of event-slots, where scripts can be attached and run when the event is fired by the game engine. In this dissertation, potential conflicts only arise when code snippets generated from different patterns are inserted into the same script, therefore to detect potential conflicts I organized patterns based on the objects and event-slots they are associated with. The ScriptEase Conflict Advisor provides this organization by present the alternative view – *Object-slot View*. Figure 3.6 illustrates the concept of the Object-slot View in a tree form similar to the one seen in Figure 3.5.

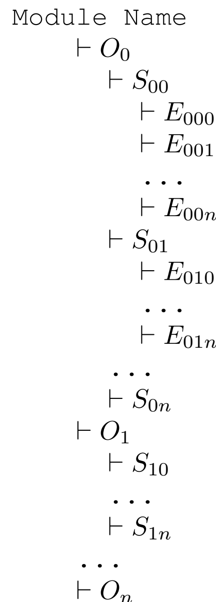

Figure 3.6: Tree view in the Object-slot View

In Figure 3.6, "O" represents a game object and "S" represents an event-slot. This object-slot tree uses a three-level hierarchy and the three levels are (from root to leaf):

- Level 1 – game objects;

- Level 2 – event-slots of its game object parent;
- Level 3 – patterns attached to the event-slot parent of the game object grandparent.

The first two subscripts of a tree node indicate the tree node's parent and grandparent.

The Object-slot View categorizes all the patterns in a module based on their associated game objects and event-slots, then displays them in a tree as shown in Figure 3.6. The patterns in Figure 3.6 can all be found in Figure 3.5, since the Object-slot View is merely an organizer that takes all the patterns displayed in the Pattern Builder and represents them in an alternative form. Unlike the Pattern builder, the user has no control over the tree in the Object-slot View and movement of tree nodes is not allowed. This is because this view is derived from properties of created patterns.

It should also be noticed that patterns appearing in the Object-slot View may not be unique. In the current version of ScriptEase, an encounter pattern or a dialog pattern, which is always attached to one event-slot, can only be found once in the object-slot tree. However, for a quest pattern or behaviour pattern, it is possible and common to find them occur multiple times in this tree, since these two kinds of patterns often generate code for different event-slots. For example, a *Kill* quest consists of two quest points (please refer to Section 1.6 for quest point) – one to kill a dragon and the other to talk to an NPC to obtain a reward. Thus this quest will generate scripts for at least two event-slots – one for the *OnDeath* slot of the dragon and the other for the *Action* slot of the conversation node for giving the reward. The object-slot tree containing this quest pattern will be like the one in Figure 3.7. "Q" represents a quest pattern and "Kill Dragon" is the name of the quest. The star (*) symbol represents a quest point and the two quest points in this quest are called "Kill the dragon" and "Get the reward".

```
Module Name
    ⊢ O (Dragon)
      ⊢ S (OnDeath)
        ⊢ Q (Kill Dragon)
          * (Kill the dragon)
          * (Get the reward)
    ⊢ O (Conversation Node: Here is your reward)
      ⊢ S (Action)
        ⊢ Q (Kill Dragon)
          * (Kill the dragon)
          * (Get the reward)
    ...
```

Figure 3.7: Object-slot tree of the "Kill Dragon" quest

Please notice the bold nodes in this tree. Since the *Kill Dragon* quest appears twice in the tree, there should be a way to distinguish between the two occurrences. Under the *OnDeath* slot of the *Dragon*, Quest Point *Kill the dragon* is highlighted because it is this quest point that generates the

script for the *OnDeath* slot. Analogously, Quest Point *Get the reward* is highlighted under the *Action* slot of the conversation node. This can help the user to locate the source of the generated code.

I have implemented the Object-slot View in ScriptEase. Figure 3.8 is a screenshot of this view on the module displayed in Figure 3.1 – *SETest_CastleFull.mod*.

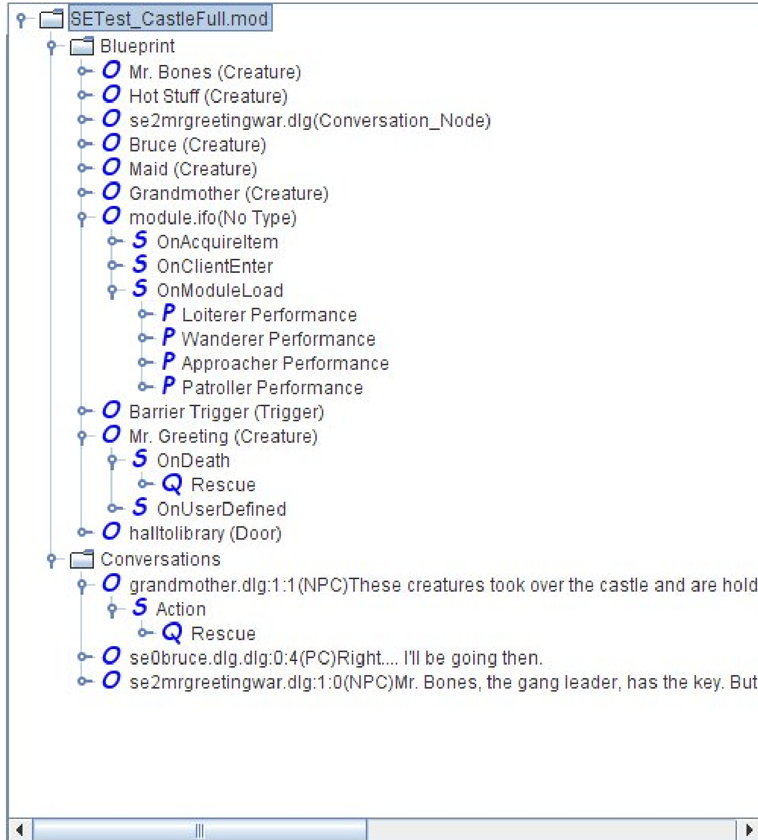

Figure 3.8: The Object-slot View in ScriptEase

The icons used in Figure 3.8 are the same as those used in the tree representation previously (e.g. "O" for object and "S" for event-slot). Icons for patterns were introduced in Section 1.6 (e.g. "P" for performance and "E" for encounter). Different from the tree representation, two additional folders can be found in this screenshot: *Blueprint* and *Conversations*. They are the two types of game objects in *NWN*, hence all game objects in the module are sorted into these two folders to help users find these objects by category. More work has been done to filter game objects and will be introduced in Section 3.2.4.

3.2.3 Pattern Builder vs. Object-slot View

Both Figure 3.1 and Figure 3.8 show patterns of the module *SETest_CastleFull* in a tree view, but the information conveyed is quite different. By looking at the Pattern Builder in Figure 3.1, we instantly know that there are 8 patterns in this module and their types (e.g. quest or encounter). This information is not so obvious when we look at the Object-slot View in Figure 3.8, where patterns are

no longer first-level nodes in the tree. However, Figure 3.8 provides a quick summary of all scripted objects and to which event-slots the scripts are attached. By further expanding the tree, we can discover which patterns contribute to the script of one specific event-slot in order to locate potential conflicts. For example, in Figure 3.8, under the *OnModuleLoad* slot of the *module.ifo* object (which is the module itself, but should also be considered a game object since scripts can be attached to it), there are four patterns – *Loiterer Performance*, *Wanderer Performance*, *Approacher Performance* and *Patroller Performance*. These four patterns are generating code for the same event-slot and should be examined by the user to see if potential conflicts would occur.

Since conflicts could only arise when multiple patterns are generating scripts to the same event-slot, it would be convenient to show only the object and slot nodes with more than one pattern in the tree. This has been implemented in the Conflict Advisor and once this "show conflict only" switch is toggled, the Object-slot View will be like Figure 3.9. All other objects and event-slots that contain less than two patterns have been filtered. With this feature, the user can look for potential conflicts among a few candidates. For example, in Figure 3.9, only *OnModuleLoad* of *module.ifo* and *OnUserDefined* of *Mr.Greeting* are possible candidates for detecting conflicts.

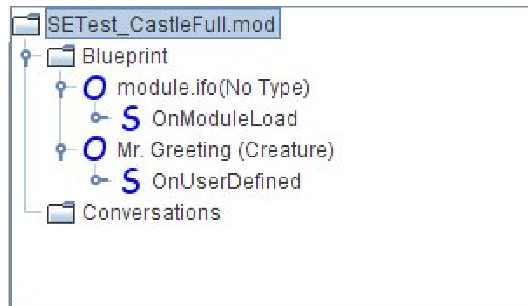

Figure 3.9: The Object-slot View with "show conflict only" turned on

3.2.4 Filtering Game Objects

Although the object-slot tree is tailored for presenting a quick overview of object-slot information, the tree will grow enormously as the number of scripted game objects increases, resulting in content overload. To address this issue, an object filter has been developed. The prototype is shown in Figure 3.10. There are four main categories of game objects in this prototype: Palette, Areas, Conversations and Module, corresponding to each tab on the top, plus the *All* tab, which shows all objects (i.e. no filter is applied). Selecting the Palette Tab reveals a collection of icons representing game objects such as creature, door, placeable, trigger, etc. The GUI of this object filter tool is borrowed from the ScriptEase picker tool. For example, the Palette is similar to the Blueprint Picker shown in Figure 1.1. Once the user is familiar with ScriptEase, this filter GUI will be easy to understand.

When the user filters objects, the Object-slot View will only show a subtree that corresponds to

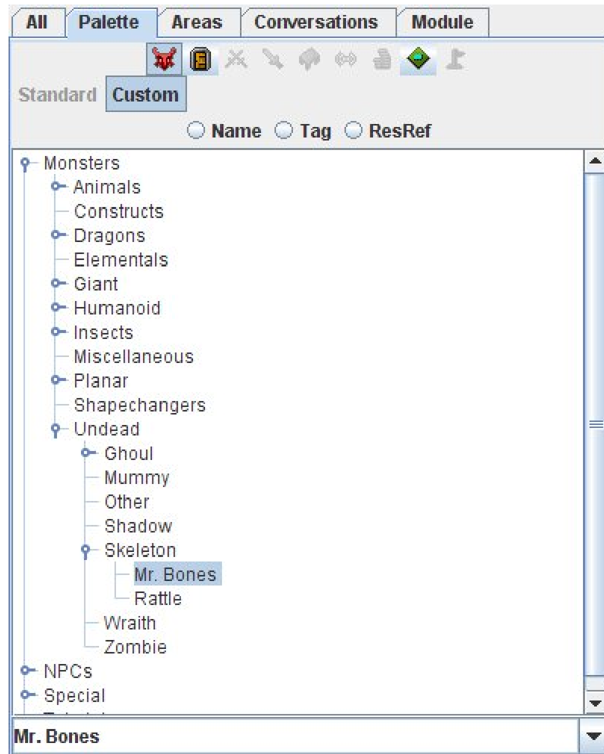

Figure 3.10: Object Filter for Conflict Advisor

the selected object(s). In Figure 3.11, *Bruce* is selected from the Palette and only this node and its children are displayed in the object-slot tree on the right. This figure also presents the entire GUI of the ScriptEase Conflict Advisor.

3.3 Summary

Now let's revisit the three requirements for the Conflict Advisor to fulfill the inadequacies of the ScriptEase Pattern Builder and the Code Viewer discussed in Section 3.2. First, the Conflict Advisor offers the Object-slot View, a tree view *graphical user interface*. Second, the organization of the object-slot tree in this view is custom-made to *locate potentially-conflicting patterns*. The organization is directly related to event-based code generation. Third, the Conflict Advisor is able to *suggest conflicts* by displaying all potentially-conflicting patterns in a module, with additional features such as showing only conflict candidates and filtering game objects.

The final chapter will present the conclusion of this thesis and a discussion of future work.

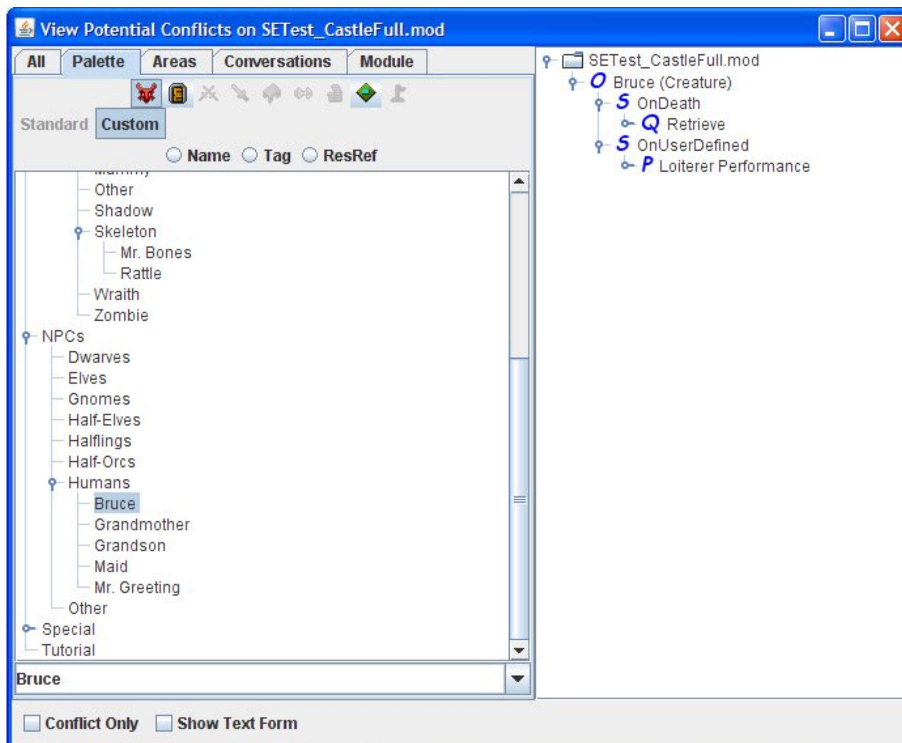

Figure 3.11: ScriptEase Conflict Advisor

Chapter 4

Conclusion and Future Work

Scripting languages have been widely employed in video games to create content for interactive story-telling. Game designers, who used to write in scripting languages, are now shifting to code-generating tools, because these tools do not require programming knowledge and they enable designers to focus on the story, leaving technical details to the tools. However, for event-based games, code generation from independent sources may cause potential inefficiencies and conflicts. This thesis proposed solutions to both issues. The code optimization was devised to reduce inefficiencies by eliminating various redundancies in the generated scripts. The Conflict Advisor was designed to assist game designers to detect potential conflicts by offering the tree-based Object-slot View. Both have been implemented in ScriptEase, a code-generating software for story-telling video games, developed at the Department of Computing Science, University of Alberta.

The code optimization incorporated traditional compiler optimization techniques with an original approach to eliminate redundant code generated from independent sources. Experiments showed an average reduction rate of 25% on ScriptEase-generated NWScript and 14% on the byte-codes compiled from these scripts. These results demonstrated the feasiblity of optimizing auto-generated scripts for event-based computer games and that optimization should be employed in tools similar to ScriptEase to achieve improved performance of scripting language execution. An immediate direction for future work is to extend the applicability of my optimization to areas beyond video games. For example, *aspect-oriented programming* (AOP) [16] may be a good candidate, since in AOP, code weavers may insert code snippets generated from different aspects into the same join point, which is similar to event-based code generation from independent sources discussed in this thesis. Unfortunately, after looking into this field, I realized many assumptions made in my thesis (e.g. I have knowledge of whether an API call has side-effects or not) were not valid in AOP and there were too many restrictions to make my optimization useful. Fuuture exploration for other applicable candidates may be beneficial.

The Conflict Advisor was created in order to provide assistance to game designers in identifying potential conflicts introduced by ScriptEase when generating scripts for the same event-slot. Since neither the ScriptEase Pattern Builder nor the Code Viewer is suitable for this task, the Object-slot

View was developed to overcome the inadequacies of the two former tools. By presenting patterns under the object-slot for which they generate code, this new view can convey information directly related to code generation and suggest potential conflicts among patterns of all types. This idea of advising game designers of the potentially-conflicted locations can lead to a new research direction: if we could look into the code marked as potentially conflicting by the Conflict Advisor and locate exactly which expressions or function calls are interfering with each other, the designers would have even more detailed feedback, since the the scope of the conflict would be reduced from an entire encounter to a single action. This would require parsing the generated scripts and inspecting data flow in the code, which has been partially done in the code optimization performed to support this research. However, more extensive analysis would be needed to perform precise conflict detection at the code level.

In this thesis, I have presented solutions to potential inefficiencies and conflicts in event-based code generation from independent sources – the code optimization to reduce the former and the Conflict Advisor to detect the latter. The code optimization is able to eliminate redundancies in ScriptEase-generated code; both raw scripts and compiled byte-codes can be shortened. The Conflict Advisor provides an alternative view which presents game objects, event-slots and ScriptEase patterns in a way that is convenient for game designers to find unintended conflicts.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] Paul Biggar, Edsko de Vries, and David Gregg. A practical solution for scripting language compilers. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied computing*, pages 1916–1923, New York, NY, USA, 2009. ACM.

[3] BioWare. `http://www.bioware.com`, 1995.

[4] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 159–170, New York, NY, USA, 1994. ACM.

[5] F. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[6] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on ssa form. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 273–286, New York, NY, USA, 1997. ACM.

[7] Keith Cooper, Jason Eckhardt, and Ken Kennedy. Redundancy elimination revisited. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 12–21, New York, NY, USA, 2008. ACM.

[8] Maria Cutumisu, Curtis Onuczko, Matthew McNaughton, Thomas Roy, Jonathan Schaeffer, Allan Schumacher, Jeff Siegel, Duane Szafron, Kevin Waugh, Mike Carbonaro, Harvey Duff, and Stephanie Gillis. Scriptease: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 67(1):32–58, 2007.

[9] Maria Cutumisu and Duane Szafron. An architecture for game behavior ai: Behavior multi-queues. *AIIDE-09*, 2009.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming Methods, Tools, and Applications (Chapter 11 Intentional Programming)*. Addison-Wesley, Reading, MA, 2000.

[11] Edsger Dijkstra. Go to statement considered harmful. *Communications of the ACM 11*, pages 147–148, 1968.

[12] Edsger W. Dijkstra. Chapter i: Notes on structured programming. pages 1–82, 1972.

[13] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting A Compiler*. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1988.

[14] Google Web Toolkit. `http://code.google.com/webtoolkit/`, 2006.

[15] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in ssa form. *ACM Trans. Program. Lang. Syst.*, 21(3):627–676, 1999.

[16] G.J. Kiczales, C. Boyapati, E.A. Hilsdale, J.J. Hugunin, C.V. Lopes, and J.O. Lamping. Aspect-Oriented Programming. 2000.

[17] Nick Lindridge. *The PHP Accelerator 1.2*, 2002.

[18] Lua. `http://www.lua.org`.

[19] M. MacNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. Scriptease: Generative design patterns for computer role-playing games. In *19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 88–99, Linz, Austria, September 2004.

[20] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, 1979.

[21] NWN Lexicon. `http://www.nwnlexicon.com/`.

[22] Neverwinter Nights. `http://nwn.bioware.com`, 2002.

[23] Curtis Onuczko, Maria Cutumisu, Duane Szafron, Jonathan Schaeffer, Matthew McNaughton, Thomas Roy, Kevin Waugh, Mike Carbonaro, and Jeff Siegel. A pattern catalog for computer role playing games. In *GameOn '05 North America*, pages 33–38, Montreal, Canada, August 2005.

[24] PHP. `http://www.php.net`.

[25] PHP Accelerator. `http://www.php-accelerator.co.uk/`.

[26] Python. `http://www.python.org`.

[27] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, Inc., 1994.

[28] ScriptEase. `http://www.cs.ualberta.ca/~script`, 2004.

[29] Marcus Alexander Trenton. Quest patterns for story-based video games. Master's thesis, University of Alberta, 2009.

[30] Warcraft III. `http://www.blizzard.com/us/war3`, 2002.

[31] Zend Optimizer. `http://www.zend.com/store/products/zend-optimizer.php`.

Appendix A

Code from *The Summer Story – Phase 1*

List of Code

A.1 scriptease1.nss before optimization 53
A.2 scriptease1.nss after optimization 56
A.3 scriptease2.nss before optimization 60
A.4 scriptease2.nss after optimization 62
A.5 scriptease3.nss before optimization 64
A.6 scriptease3.nss after optimization 67
A.7 scriptease4.nss before optimization 72
A.8 scriptease4.nss after optimization 74
A.9 scriptease5.nss before optimization 77
A.10 scriptease5.nss after optimization 79
A.11 scriptease6.nss before optimization 81
A.12 scriptease6.nss after optimization 83
A.13 scriptease7.nss before optimization 85
A.14 scriptease7.nss after optimization 87
A.15 scriptease8.nss before optimization 89
A.16 scriptease8.nss after optimization 91

Code A.1: scriptease1.nss before optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
void SE_Ac_JumpTowardsObject(object param_1, object param_2, float param_3);
void SE_Ac_ShowImpactVisualEffectOnUsableObject(int param_1, object param_2);
int SE_Df_ItemEquipped(object param_1, object param_2, int param_3);
void SE_Ac_JumpTowardsObject(object param_1, object param_2, float param_3) {
  location jumperLocation;
  vector jumperPosition;
  float jumperFacing;
  object jumperArea;
  vector targetPosition;
  vector heading;

  jumperLocation = GetLocation(param_1);
  jumperPosition = GetPositionFromLocation(jumperLocation);
  jumperArea = GetAreaFromLocation(jumperLocation);
  jumperFacing = GetFacingFromLocation(jumperLocation);
  targetPosition = GetPositionFromLocation(GetLocation(param_2));
  heading = Vector(targetPosition.x - jumperPosition.x, targetPosition.y -
    jumperPosition.y, targetPosition.z - jumperPosition.z);
  heading = VectorNormalize(heading);
  jumperPosition = jumperPosition + param_3*heading;
  jumperLocation = Location(jumperArea, jumperPosition, jumperFacing);
  AssignCommand(param_1, JumpToLocation(jumperLocation));
}

void SE_Ac_ShowImpactVisualEffectOnUsableObject(int param_1, object param_2) {
  effect veffect = EffectVisualEffect(param_1);
  ApplyEffectToObject(DURATION_TYPE_INSTANT, veffect, param_2);
}

int SE_Df_ItemEquipped(object param_1, object param_2, int param_3) {
  if (param_2 == OBJECT_INVALID)
    return FALSE;
  else
    return GetItemInSlot(param_3, param_1) == param_2;
}

//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



void TriggerenterbarrierspecificitemTrytoentertrigger_1();

void TriggerenterbarrierspecificitemTrytoentertrigger_1() {

  // The following are all of the variables used in this situation

  int IsEquipped_SE4;
  int SameObjects_SE6;
  object ThePC_SE5;
  float NegativeBounceDistance_SE2;
```

```
object UnderwaterBreathingHelmetandMechanism_SE3;
object DeadCentreBounce_SE7;
object TheTrigger_SE1;
object Enterer_SE0;


// When a creature enters DeadInWater
if( ! TRUE ) return;

// Define Negative Bounce Distance as 2.5 negated
NegativeBounceDistance_SE2 = -2.5;

// Get the object with tag "UnderwaterBreathingHelmetandMech"
UnderwaterBreathingHelmetandMechanism_SE3 =
  GetNearestObjectByTag("UnderwaterBreathingHelmetandMech");
if (UnderwaterBreathingHelmetandMechanism_SE3 == OBJECT_INVALID )
  UnderwaterBreathingHelmetandMechanism_SE3 =
    GetObjectByTag("UnderwaterBreathingHelmetandMech");

// Define Is Equipped as whether Enterer has Underwater Breathing Helmet
// and Mechanism in Head
IsEquipped_SE4 = SE_Df_ItemEquipped(Enterer_SE0,
  UnderwaterBreathingHelmetandMechanism_SE3,
  INVENTORY_SLOT_HEAD);

// Define The PC as the PC
ThePC_SE5 = GetFirstPC();

// Define Same Objects as whether The PC is the same as Enterer
SameObjects_SE6 = ThePC_SE5 == Enterer_SE0;

// Define Enterer as the creature that just entered DeadInWater
Enterer_SE0 = GetEnteringObject();

// Define The Trigger as the object the event is fired on
TheTrigger_SE1 = OBJECT_SELF;

// Define Negative Bounce Distance as 2.5 negated
NegativeBounceDistance_SE2 = -2.5;

// Define Is Equipped as whether Enterer has Underwater Breathing Helmet
// and Mechanism in Head
IsEquipped_SE4 = SE_Df_ItemEquipped(Enterer_SE0,
  UnderwaterBreathingHelmetandMechanism_SE3,
  INVENTORY_SLOT_HEAD);

// Define The PC as the PC
ThePC_SE5 = GetFirstPC();

// Define Same Objects as whether The PC is the same as Enterer
SameObjects_SE6 = ThePC_SE5 == Enterer_SE0;


// Main code body - checks conditions and executes actions

// Sets up the multiCondition with selected BinaryDefinitions
if( !IsEquipped_SE4 && SameObjects_SE6 ) {

  // Get the object with tag "DeadCentreBounce"
```

```
    DeadCentreBounce_SE7 = GetNearestObjectByTag("DeadCentreBounce");
    if (DeadCentreBounce_SE7 == OBJECT_INVALID )
      DeadCentreBounce_SE7 = GetObjectByTag("DeadCentreBounce");

    // Enterer jumps towards DeadCentreBounce, a distance Negative Bounce Distance with
    // visual effect Confusion
    // Show the Confusion impact visual effect on Enterer
    SE_Ac_ShowImpactVisualEffectOnUsableObject(VFX_IMP_CONFUSION_S, Enterer_SE0);
    // Enterer jumps Negative Bounce Distance towards DeadCentreBounce
    SE_Ac_JumpTowardsObject(Enterer_SE0, DeadCentreBounce_SE7,
      NegativeBounceDistance_SE2);

    // Show  Going any further will cause death. You must turn back. above Enterer
    FloatingTextStringOnCreature(
      "Going␣any␣further␣will␣cause␣death.␣You␣must␣turn␣back.",
      Enterer_SE0 );

  }

}

//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {
//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  TriggerenterbarrierspecificitemTrytoentertrigger_1();
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.2: scriptease1.nss after optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
//@Optimization performed: function prototype declaration removed
//void SE_Ac_JumpTowardsObject(object param_1, object param_2, float param_3);
//@Optimization performed: function prototype declaration removed
//void SE_Ac_ShowImpactVisualEffectOnUsableObject(int param_1, object param_2);
//@Optimization performed: function prototype declaration removed
//int SE_Df_ItemEquipped(object param_1, object param_2, int param_3);
//@Optimization performed: function inlining, original declaration removed

//@Optimization performed: function inlining, original declaration removed

//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



//@Optimization performed: function prototype declaration removed
//void TriggerenterbarrierspecificitemTrytoentertrigger_1();
//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {

//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  //@Optimization performed: function inlining
  // (TriggerenterbarrierspecificitemTrytoentertrigger_1)


  // The following are all of the variables used in this situation

  int IsEquipped_SE4;
  int SameObjects_SE6;
  object ThePC_SE5;
  float NegativeBounceDistance_SE2;
  object UnderwaterBreathingHelmetandMechanism_SE3;
  object DeadCentreBounce_SE7;
  object TheTrigger_SE1;
  object Enterer_SE0;
```

```
  //@ptimization performed: dead branch removed
  /*


  // When a creature enters DeadInWater
  if(*//*@Optimization performed: constant pre-computation*//*FALSE ) return;*/

  // Define Negative Bounce Distance as 2.5 negated
  NegativeBounceDistance_SE2 = -2.5;

  // Get the object with tag "UnderwaterBreathingHelmetandMech"
  UnderwaterBreathingHelmetandMechanism_SE3 =
    GetNearestObjectByTag("UnderwaterBreathingHelmetandMech");
  if (UnderwaterBreathingHelmetandMechanism_SE3 == OBJECT_INVALID )
    UnderwaterBreathingHelmetandMechanism_SE3 =
      GetObjectByTag("UnderwaterBreathingHelmetandMech");
 //@Optimization performed: function inlining (SE_Df_ItemEquipped)

int SE_Df_ItemEquipped_0;
  if (UnderwaterBreathingHelmetandMechanism_SE3 == OBJECT_INVALID)
    SE_Df_ItemEquipped_0 = FALSE;
  else
    SE_Df_ItemEquipped_0 = GetItemInSlot(INVENTORY_SLOT_HEAD, Enterer_SE0) ==
      UnderwaterBreathingHelmetandMechanism_SE3;

  // Define Is Equipped as whether Enterer has Underwater Breathing Helmet
  // and Mechanism in Head
  IsEquipped_SE4 = SE_Df_ItemEquipped_0;

  // Define The PC as the PC
  ThePC_SE5 = GetFirstPC();

  // Define Same Objects as whether The PC is the same as Enterer
  SameObjects_SE6 = ThePC_SE5 == Enterer_SE0;

  // Define Enterer as the creature that just entered DeadInWater
  Enterer_SE0 = GetEnteringObject();

  // Define The Trigger as the object the event is fired on
  TheTrigger_SE1 = OBJECT_SELF;
  //@Optimization performed: redundant assignment removed.
/*

  // Define Negative Bounce Distance as 2.5 negated
  NegativeBounceDistance_SE2 = -2.5;*/
 //@Optimization performed: function inlining (SE_Df_ItemEquipped)

int SE_Df_ItemEquipped_1;
  if (UnderwaterBreathingHelmetandMechanism_SE3 == OBJECT_INVALID)
    SE_Df_ItemEquipped_1 = FALSE;
  else
    SE_Df_ItemEquipped_1 = GetItemInSlot(INVENTORY_SLOT_HEAD, Enterer_SE0) ==
      UnderwaterBreathingHelmetandMechanism_SE3;

  // Define Is Equipped as whether Enterer has Underwater Breathing Helmet
  // and Mechanism in Head
  IsEquipped_SE4 = SE_Df_ItemEquipped_1;
  //@Optimization performed: redundant assignment removed.
/*
```

```
  // Define The PC as the PC
  ThePC_SE5 = GetFirstPC();*/
  //@Optimization performed: redundant assignment removed.
/*

  // Define Same Objects as whether The PC is the same as Enterer
  SameObjects_SE6 = ThePC_SE5 == Enterer_SE0;*/


  // Main code body - checks conditions and executes actions

  // Sets up the multiCondition with selected BinaryDefinitions
  if( !IsEquipped_SE4 && SameObjects_SE6 ) {

    // Get the object with tag "DeadCentreBounce"
    DeadCentreBounce_SE7 = GetNearestObjectByTag("DeadCentreBounce");
    if (DeadCentreBounce_SE7 == OBJECT_INVALID )
      DeadCentreBounce_SE7 = GetObjectByTag("DeadCentreBounce");


    // Enterer jumps towards DeadCentreBounce, a distance Negative Bounce Distance
    // with visual effect Confusion
    // Show the Confusion impact visual effect on Enterer
    //@Optimization performed: function inlining
    // (SE_Ac_ShowImpactVisualEffectOnUsableObject)

  effect veffect = EffectVisualEffect(VFX_IMP_CONFUSION_S);
  ApplyEffectToObject(DURATION_TYPE_INSTANT, veffect, Enterer_SE0);

    // Enterer jumps Negative Bounce Distance towards DeadCentreBounce
    //@Optimization performed: function inlining (SE_Ac_JumpTowardsObject)

  location jumperLocation;
  vector jumperPosition;
  float jumperFacing;
  object jumperArea;
  vector targetPosition;
  vector heading;

  jumperLocation = GetLocation(Enterer_SE0);
  jumperPosition = GetPositionFromLocation(jumperLocation);
  jumperArea = GetAreaFromLocation(jumperLocation);
  jumperFacing = GetFacingFromLocation(jumperLocation);
  targetPosition = GetPositionFromLocation(GetLocation(DeadCentreBounce_SE7));
  heading = Vector(targetPosition.x - jumperPosition.x, targetPosition.y -
    jumperPosition.y, targetPosition.z - jumperPosition.z);
  heading = VectorNormalize(heading);
  jumperPosition = jumperPosition + NegativeBounceDistance_SE2*heading;
  jumperLocation = Location(jumperArea, jumperPosition, jumperFacing);
  AssignCommand(Enterer_SE0, JumpToLocation(jumperLocation));

    // Show  Going any further will cause death. You must turn back. above Enterer
    FloatingTextStringOnCreature(
      "Going_any_further_will_cause_death._You_must_turn_back.",
      Enterer_SE0 );

  }
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
```

```
// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.3: scriptease2.nss before optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



void AreaenterAreaenteredaddactions_2();

void AreaenterAreaenteredaddactions_2() {

  // The following are all of the variables used in this situation

  object AreaEnterer_SE1;
  object Doorway_SE2;
  object wemarketare_SE0;

  // This script is attached to the following object's OnEnter script slot
  wemarketare_SE0 = OBJECT_SELF;

  // When a creature enters we_market.are
  if( ! TRUE ) return;

  // Define Area Enterer as the creature that just entered we_market.are
  AreaEnterer_SE1 = GetEnteringObject();


  // Main code body - checks conditions and executes actions

  // Get the object with tag "WharehouseEntrance"
  Doorway_SE2 = GetNearestObjectByTag("WharehouseEntrance");
  if (Doorway_SE2 == OBJECT_INVALID )
    Doorway_SE2 = GetObjectByTag("WharehouseEntrance");

  // Animate Doorway to Close
  AssignCommand(Doorway_SE2, PlayAnimation(ANIMATION_PLACEABLE_CLOSE));


}

//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {
//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
```

```
  AreaenterAreaenteredaddactions_2();
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.4: scriptease2.nss after optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



//@Optimization performed: function prototype declaration removed
//void AreaenterAreaenteredaddactions_2();
//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {

//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  //@Optimization performed: function inlining (AreaenterAreaenteredaddactions_2)


  // The following are all of the variables used in this situation

  object AreaEnterer_SE1;
  object Doorway_SE2;
  object wemarketare_SE0;

  // This script is attached to the following object's OnEnter script slot
  wemarketare_SE0 = OBJECT_SELF;
  //@ptimization performed: dead branch removed
  /*

  // When a creature enters we_market.are
  if(*//*@Optimization performed: constant pre-computation*//*FALSE ) return;*/

  // Define Area Enterer as the creature that just entered we_market.are
  AreaEnterer_SE1 = GetEnteringObject();


  // Main code body - checks conditions and executes actions

  // Get the object with tag "WharehouseEntrance"
  Doorway_SE2 = GetNearestObjectByTag("WharehouseEntrance");
  if (Doorway_SE2 == OBJECT_INVALID )
    Doorway_SE2 = GetObjectByTag("WharehouseEntrance");
```

```
  // Animate Doorway to Close
  AssignCommand(Doorway_SE2, PlayAnimation(ANIMATION_PLACEABLE_CLOSE));
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.5: scriptease3.nss before optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
int SE_Co_IsNegative(int param_1);
int SE_Co_IsPositive(int param_1);
string SE_Df_GetCurrentGameTimeHM();
int SE_Co_IsNegative(int param_1) {
  return  ! param_1;
}

int SE_Co_IsPositive(int param_1) {
  return param_1;
}

string SE_Df_GetCurrentGameTimeHM() {
  int nHour = GetTimeHour();
  string sHour = IntToString(nHour);
  if (nHour <= 9) sHour = "0" + sHour;

  int nMinute = GetTimeMinute();
  string sMinute = IntToString(nMinute);
  if (nMinute <= 9) sMinute = "0" + sMinute;

  string sTime = sHour + ":" + sMinute;
  return sTime;
}

//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



void PlaceableuseUseplaceableaddactions_3();
void PlaceableuseUseplaceableaddactions_9();

void PlaceableuseUseplaceableaddactions_3() {

  // The following are all of the variables used in this situation

  string Time_SE2;
  object ThePlaceable_SE1;
  int IsNighttime_SE3;
  object User_SE0;


  // When Sundial is used
  if( ! TRUE ) return;

  // Define Time as the current game time (hh:mm)
  Time_SE2 = SE_Df_GetCurrentGameTimeHM();

  // Define Is Nighttime as whether it is nighttime
  IsNighttime_SE3 = GetIsNight();
```

```
  // Define User as the user of Sundial
  User_SE0 = GetLastUsedBy();

  // Define The Placeable as the object the event is fired on
  ThePlaceable_SE1 = OBJECT_SELF;

  // Define Time as the current game time (hh:mm)
  Time_SE2 = SE_Df_GetCurrentGameTimeHM();

  // Define Is Nighttime as whether it is nighttime
  IsNighttime_SE3 = GetIsNight();


  // Main code body - checks conditions and executes actions

  // Do the following actions if Is Nighttime is false
  // If Is Nighttime is Negative (False, No, Off, etc.)
  if( SE_Co_IsNegative(IsNighttime_SE3) ) {
    // The Placeable speaks Time
    AssignCommand(ThePlaceable_SE1, SpeakString(Time_SE2, TALKVOLUME_TALK));
  }

  // Do the following actions if Is Nighttime is true
  // If Is Nighttime is Positive (True, Yes, On, etc.)
  if( SE_Co_IsPositive(IsNighttime_SE3) ) {
    // The Placeable speaks Sundials do not function at night.
    AssignCommand(ThePlaceable_SE1, SpeakString("Sundials␣do␣not␣function␣at␣night.",
      TALKVOLUME_TALK));
  }


}

void PlaceableuseUseplaceableaddactions_9() {

  // The following are all of the variables used in this situation

  string Time_SE2;
  int IsNighttime_SE3;
  object ThePlaceable_SE1;
  object User_SE0;


  // When Sundial is used
  if( ! TRUE ) return;

  // Define Time as the current game time (hh:mm)
  Time_SE2 = SE_Df_GetCurrentGameTimeHM();

  // Define Is Nighttime as whether it is nighttime
  IsNighttime_SE3 = GetIsNight();

  // Define User as the user of Sundial
  User_SE0 = GetLastUsedBy();

  // Define The Placeable as the object the event is fired on
  ThePlaceable_SE1 = OBJECT_SELF;

  // Define Time as the current game time (hh:mm)
```

```
  Time_SE2 = SE_Df_GetCurrentGameTimeHM();

  // Define Is Nighttime as whether it is nighttime
  IsNighttime_SE3 = GetIsNight();


  // Main code body - checks conditions and executes actions

  // Do the following actions if Is Nighttime is false
  // If Is Nighttime is Negative (False, No, Off, etc.)
  if( SE_Co_IsNegative(IsNighttime_SE3) ) {
    // The Placeable speaks Time
    AssignCommand(ThePlaceable_SE1, SpeakString(Time_SE2, TALKVOLUME_TALK));
  }

  // Do the following actions if Is Nighttime is true
  // If Is Nighttime is Positive (True, Yes, On, etc.)
  if( SE_Co_IsPositive(IsNighttime_SE3) ) {
    // The Placeable speaks Sundials do not function at night.
    AssignCommand(ThePlaceable_SE1, SpeakString("Sundials do not function at night.",
      TALKVOLUME_TALK));
  }


}

//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {
//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  PlaceableuseUseplaceableaddactions_3();
  PlaceableuseUseplaceableaddactions_9();
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.6: scriptease3.nss after optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
//@Optimization performed: function prototype declaration removed
//int SE_Co_IsNegative(int param_1);
//@Optimization performed: function prototype declaration removed
//int SE_Co_IsPositive(int param_1);
//@Optimization performed: function prototype declaration removed
//string SE_Df_GetCurrentGameTimeHM();
//@Optimization performed: function inlining, original declaration removed

//@Optimization performed: function inlining, original declaration removed

//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



//@Optimization performed: function prototype declaration removed
//void PlaceableuseUseplaceableaddactions_3();
//@Optimization performed: function prototype declaration removed
//void PlaceableuseUseplaceableaddactions_9();
//@Optimization performed: function inlining, original declaration removed

//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {

//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  //@Optimization performed: function inlining (PlaceableuseUseplaceableaddactions_3)


  // The following are all of the variables used in this situation

  string Time_SE2;
  object ThePlaceable_SE1;
  int IsNighttime_SE3;
  object User_SE0;
  //@ptimization performed: dead branch removed
```

```
  /*


  // When Sundial is used
  if(*//*@Optimization performed: constant pre-computation*//*FALSE ) return;*/
 //@Optimization performed: function inlining (SE_Df_GetCurrentGameTimeHM)

string SE_Df_GetCurrentGameTimeHM_0;
  int nHour = GetTimeHour();
  string sHour = IntToString(nHour);
  if (nHour <= 9) sHour = "0" + sHour;

  int nMinute = GetTimeMinute();
  string sMinute = IntToString(nMinute);
  if (nMinute <= 9) sMinute = "0" + sMinute;

  string sTime = sHour + ":" + sMinute;
  SE_Df_GetCurrentGameTimeHM_0 = sTime;

  // Define Time as the current game time (hh:mm)
  Time_SE2 = SE_Df_GetCurrentGameTimeHM_0;

  // Define Is Nighttime as whether it is nighttime
  IsNighttime_SE3 = GetIsNight();

  // Define User as the user of Sundial
  User_SE0 = GetLastUsedBy();

  // Define The Placeable as the object the event is fired on
  ThePlaceable_SE1 = OBJECT_SELF;
 //@Optimization performed: function inlining (SE_Df_GetCurrentGameTimeHM)

string SE_Df_GetCurrentGameTimeHM_1;
  //@Optimization performed: redundant variable declaration removed
/*
  int nHour = GetTimeHour();*/
  //@Optimization performed: redundant variable declaration removed
/*
  string sHour = IntToString(nHour);*/
  if (nHour <= 9)
                //@Optimization performed: redundant assignment removed.
/* sHour = "0" + sHour;*/
  //@Optimization performed: redundant variable declaration removed
/*

  int nMinute = GetTimeMinute();*/
  //@Optimization performed: redundant variable declaration removed
/*
  string sMinute = IntToString(nMinute);*/
  if (nMinute <= 9)
                  //@Optimization performed: redundant assignment removed.
/* sMinute = "0" + sMinute;*/
  //@Optimization performed: redundant variable declaration removed
/*

  string sTime = sHour + ":" + sMinute;*/
  SE_Df_GetCurrentGameTimeHM_1 = sTime;

  // Define Time as the current game time (hh:mm)
```

```
  Time_SE2 = SE_Df_GetCurrentGameTimeHM_1;
  //@Optimization performed: redundant assignment removed.
/*

  // Define Is Nighttime as whether it is nighttime
  IsNighttime_SE3 = GetIsNight();*/
 //@Optimization performed: function inlining (SE_Co_IsNegative)

int SE_Co_IsNegative_2;
  SE_Co_IsNegative_2 =  ! IsNighttime_SE3;


  // Main code body - checks conditions and executes actions

  // Do the following actions if Is Nighttime is false
  // If Is Nighttime is Negative (False, No, Off, etc.)
  if( SE_Co_IsNegative_2 ) {
    // The Placeable speaks Time
    AssignCommand(ThePlaceable_SE1, SpeakString(Time_SE2, TALKVOLUME_TALK));
  }
 //@Optimization performed: function inlining (SE_Co_IsPositive)

int SE_Co_IsPositive_3;
  SE_Co_IsPositive_3 = IsNighttime_SE3;

  // Do the following actions if Is Nighttime is true
  // If Is Nighttime is Positive (True, Yes, On, etc.)
  if( SE_Co_IsPositive_3 ) {
    // The Placeable speaks Sundials do not function at night.
    AssignCommand(ThePlaceable_SE1, SpeakString("Sundials do not function at night.",
      TALKVOLUME_TALK));
  }

  //@Optimization performed: function inlining (PlaceableuseUseplaceableaddactions_9)

  //@Optimization performed: redundant variable declaration removed
/*

  // The following are all of the variables used in this situation

  string Time_SE2;*/
  //@Optimization performed: redundant variable declaration removed
/*
  int IsNighttime_SE3;*/
  //@Optimization performed: redundant variable declaration removed
/*
  object ThePlaceable_SE1;*/
  //@Optimization performed: redundant variable declaration removed
/*
  object User_SE0;*/
  //@ptimization performed: dead branch removed
  /*


  // When Sundial is used
  if(*//*@Optimization performed: constant pre-computation*//*FALSE ) return;*/
 //@Optimization performed: function inlining (SE_Df_GetCurrentGameTimeHM)

string SE_Df_GetCurrentGameTimeHM_4;
```

```
  //@Optimization performed: redundant variable declaration removed
/*
  int nHour = GetTimeHour();*/
  //@Optimization performed: redundant variable declaration removed
/*
  string sHour = IntToString(nHour);*/
  if (nHour <= 9) sHour = "0" + sHour;
  //@Optimization performed: redundant variable declaration removed
/*

  int nMinute = GetTimeMinute();*/
  //@Optimization performed: redundant variable declaration removed
/*
  string sMinute = IntToString(nMinute);*/
  if (nMinute <= 9) sMinute = "0" + sMinute;
  //@Optimization performed: redundant variable declaration removed
/*

  string sTime = sHour + ":" + sMinute;*/
  SE_Df_GetCurrentGameTimeHM_4 = sTime;

  // Define Time as the current game time (hh:mm)
  Time_SE2 = SE_Df_GetCurrentGameTimeHM_4;

  // Define Is Nighttime as whether it is nighttime
  IsNighttime_SE3 = GetIsNight();
  //@Optimization performed: redundant assignment removed.
/*

  // Define User as the user of Sundial
  User_SE0 = GetLastUsedBy();*/
  //@Optimization performed: redundant assignment removed.
/*

  // Define The Placeable as the object the event is fired on
  ThePlaceable_SE1 = OBJECT_SELF;*/
 //@Optimization performed: function inlining (SE_Df_GetCurrentGameTimeHM)

string SE_Df_GetCurrentGameTimeHM_5;
  //@Optimization performed: redundant variable declaration removed
/*
  int nHour = GetTimeHour();*/
  //@Optimization performed: redundant variable declaration removed
/*
  string sHour = IntToString(nHour);*/
  if (nHour <= 9)
                  //@Optimization performed: redundant assignment removed.
/* sHour = "0" + sHour;*/
  //@Optimization performed: redundant variable declaration removed
/*

  int nMinute = GetTimeMinute();*/
  //@Optimization performed: redundant variable declaration removed
/*
  string sMinute = IntToString(nMinute);*/
  if (nMinute <= 9)
                    //@Optimization performed: redundant assignment removed.
/* sMinute = "0" + sMinute;*/
  //@Optimization performed: redundant variable declaration removed
```

```
/*

  string sTime = sHour + ":" + sMinute;*/
  SE_Df_GetCurrentGameTimeHM_5 = sTime;

  // Define Time as the current game time (hh:mm)
  Time_SE2 = SE_Df_GetCurrentGameTimeHM_5;
  //@Optimization performed: redundant assignment removed.
/*

  // Define Is Nighttime as whether it is nighttime
  IsNighttime_SE3 = GetIsNight();*/
 //@Optimization performed: function inlining (SE_Co_IsNegative)

int SE_Co_IsNegative_6;
  SE_Co_IsNegative_6 =  ! IsNighttime_SE3;


  // Main code body - checks conditions and executes actions

  // Do the following actions if Is Nighttime is false
  // If Is Nighttime is Negative (False, No, Off, etc.)
  if( SE_Co_IsNegative_6 ) {
    // The Placeable speaks Time
    AssignCommand(ThePlaceable_SE1, SpeakString(Time_SE2, TALKVOLUME_TALK));
  }
 //@Optimization performed: function inlining (SE_Co_IsPositive)

int SE_Co_IsPositive_7;
  SE_Co_IsPositive_7 = IsNighttime_SE3;

  // Do the following actions if Is Nighttime is true
  // If Is Nighttime is Positive (True, Yes, On, etc.)
  if( SE_Co_IsPositive_7 ) {
    // The Placeable speaks Sundials do not function at night.
    AssignCommand(ThePlaceable_SE1, SpeakString("Sundials do not function at night.",
      TALKVOLUME_TALK));
  }
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.7: scriptease4.nss before optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
#include "NW_I0_GENERIC"
void SE_Ac_ActionPlaceholder();
void SE_Ac_DetermineCombatRoundTodo(object param_1);
void SE_Ac_ActionPlaceholder() {

}

void SE_Ac_DetermineCombatRoundTodo(object param_1) {
  AssignCommand(param_1,
    ActionDoCommand(DetermineCombatRound())
  );
}

//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



void ConversationactionWhathappenswhenaconversationnodeisdisplayed_4();
void ConversationactionattackAttackPCspeakerwhenaconversationnodeisdisplayed_6();

void ConversationactionWhathappenswhenaconversationnodeisdisplayed_4() {

  // The following are all of the variables used in this situation

  object ObjectSpeaker_SE1;
  object NPCSpeaker_SE2;
  object PCSpeaker_SE0;


  // After amendur.dlg:0:20 is reached
  if( ! TRUE ) return;

  // Define PC Speaker as the PC speaking in the amendur.dlg:0:20
  PCSpeaker_SE0 = GetPCSpeaker();

  // Define Object Speaker as the object speaking in the amendur.dlg:0:20
  ObjectSpeaker_SE1 = OBJECT_SELF;

  // Define NPC Speaker as the NPC speaking in the amendur.dlg:0:20
  NPCSpeaker_SE2 = OBJECT_SELF;


  // Main code body - checks conditions and executes actions

  // Replace this action placeholder by one or more actions
  SE_Ac_ActionPlaceholder();


}
```

```
void ConversationactionattackAttackPCspeakerwhenaconversationnodeisdisplayed_6() {

  // The following are all of the variables used in this situation

  object ObjectSpeaker_SE1;
  object PCSpeaker_SE0;
  object NPCSpeaker_SE2;


  // After amendur.dlg:0:20 is reached
  if( ! TRUE ) return;

  // Define PC Speaker as the PC speaking in the amendur.dlg:0:20
  PCSpeaker_SE0 = GetPCSpeaker();

  // Define Object Speaker as the object speaking in the amendur.dlg:0:20
  ObjectSpeaker_SE1 = OBJECT_SELF;

  // Define NPC Speaker as the NPC speaking in the amendur.dlg:0:20
  NPCSpeaker_SE2 = OBJECT_SELF;


  // Main code body - checks conditions and executes actions

  // Make NPC Speaker attack PC Speaker
  // Set PC Speaker as an enemy of NPC Speaker for 15.0 seconds
  SetIsTemporaryEnemy(PCSpeaker_SE0, NPCSpeaker_SE2, FALSE, 15.0);
  // NPC Speaker* determines its combat round
  SE_Ac_DetermineCombatRoundTodo(NPCSpeaker_SE2);


}

//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {
//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  ConversationactionWhathappenswhenaconversationnodeisdisplayed_4();
  ConversationactionattackAttackPCspeakerwhenaconversationnodeisdisplayed_6();
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.8: scriptease4.nss after optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
#include "NW_I0_GENERIC"
//@Optimization performed: function prototype declaration removed
//void SE_Ac_ActionPlaceholder();
//@Optimization performed: function prototype declaration removed
//void SE_Ac_DetermineCombatRoundTodo(object param_1);
//@Optimization performed: function inlining, original declaration removed

//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



//@Optimization performed: function prototype declaration removed
//void ConversationactionWhathappenswhenaconversationnodeisdisplayed_4();
//@Optimization performed: function prototype declaration removed
//void ConversationactionattackAttackPCspeakerwhenaconversationnodeisdisplayed_6();
//@Optimization performed: function inlining, original declaration removed

//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {

//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  //@Optimization performed: function inlining
  // (ConversationactionWhathappenswhenaconversationnodeisdisplayed_4)


  // The following are all of the variables used in this situation

  object ObjectSpeaker_SE1;
  object NPCSpeaker_SE2;
  object PCSpeaker_SE0;
  //@ptimization performed: dead branch removed
  /*
```

```
  // After amendur.dlg:0:20 is reached
  if(*//*@Optimization performed: constant pre-computation*//*FALSE ) return;*/

  // Define PC Speaker as the PC speaking in the amendur.dlg:0:20
  PCSpeaker_SE0 = GetPCSpeaker();

  // Define Object Speaker as the object speaking in the amendur.dlg:0:20
  ObjectSpeaker_SE1 = OBJECT_SELF;

  // Define NPC Speaker as the NPC speaking in the amendur.dlg:0:20
  NPCSpeaker_SE2 = OBJECT_SELF;



  // Main code body - checks conditions and executes actions

  // Replace this action placeholder by one or more actions
  //@Optimization performed: function inlining (SE_Ac_ActionPlaceholder)


  //@Optimization performed: function inlining
  // (ConversationactionattackAttackPCspeakerwhenaconversationnodeisdisplayed_6)

  //@Optimization performed: redundant variable declaration removed
/*

  // The following are all of the variables used in this situation

  object ObjectSpeaker_SE1;*/
  //@Optimization performed: redundant variable declaration removed
/*
  object PCSpeaker_SE0;*/
  //@Optimization performed: redundant variable declaration removed
/*
  object NPCSpeaker_SE2;*/
  //@ptimization performed: dead branch removed
  /*


  // After amendur.dlg:0:20 is reached
  if(*//*@Optimization performed: constant pre-computation*//*FALSE ) return;*/
  //@Optimization performed: redundant assignment removed.
/*

  // Define PC Speaker as the PC speaking in the amendur.dlg:0:20
  PCSpeaker_SE0 = GetPCSpeaker();*/
  //@Optimization performed: redundant assignment removed.
/*

  // Define Object Speaker as the object speaking in the amendur.dlg:0:20
  ObjectSpeaker_SE1 = OBJECT_SELF;*/
  //@Optimization performed: redundant assignment removed.
/*

  // Define NPC Speaker as the NPC speaking in the amendur.dlg:0:20
  NPCSpeaker_SE2 = OBJECT_SELF;*/


  // Main code body - checks conditions and executes actions
```

```
  // Make NPC Speaker attack PC Speaker
  // Set PC Speaker as an enemy of NPC Speaker for 15.0 seconds
  SetIsTemporaryEnemy(PCSpeaker_SE0, NPCSpeaker_SE2, FALSE, 15.0);

  // NPC Speaker* determines its combat round
  //@Optimization performed: function inlining (SE_Ac_DetermineCombatRoundTodo)

  AssignCommand(NPCSpeaker_SE2,
    ActionDoCommand(DetermineCombatRound())
  );
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.9: scriptease5.nss before optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



int ConversationfilterWhenaconversationnodeisdisplayed_5();

int ConversationfilterWhenaconversationnodeisdisplayed_5() {

  // The following are all of the variables used in this situation

  object ObjectSpeaker_SE0;
  object PCSpeaker_SE1;
  object NPCSpeaker_SE2;


  // Display text for amendur.dlg:0:20 if the conditions are all positive
  if( ! TRUE ) return FALSE;

  // Define Object Speaker as the object speaking in the amendur.dlg:0:20
  ObjectSpeaker_SE0 = OBJECT_SELF;

  // Define PC Speaker as the PC speaking in the amendur.dlg:0:20
  PCSpeaker_SE1 = GetPCSpeaker();

  // Define NPC Speaker as the NPC speaking in the amendur.dlg:0:20
  NPCSpeaker_SE2 = OBJECT_SELF;


  // Main code body - checks conditions and executes actions

  // Sets up the multiCondition with selected BinaryDefinitions
  if( TRUE ) {

    /* This conversation node may be said.
     * All the conditions were true.
     */
    return TRUE;
  }

  /* Conversation node cannot be said. Some condition above
   * was false, so we fell through to here.
   */
  else
  return FALSE;
}

//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here
```

```
//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
int StartingConditional() {
//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  if( ConversationfilterWhenaconversationnodeisdisplayed_5() ) return TRUE;
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
  return FALSE;
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.10: scriptease5.nss after optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



//@Optimization performed: function prototype declaration removed
//int ConversationfilterWhenaconversationnodeisdisplayed_5();
//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
int StartingConditional() {
 //@Optimization performed: function inlining
 // (ConversationfilterWhenaconversationnodeisdisplayed_5)

int ConversationfilterWhenaconversationnodeisdisplayed_5_0;

  // The following are all of the variables used in this situation

  object ObjectSpeaker_SE0;
  object PCSpeaker_SE1;
  object NPCSpeaker_SE2;
  //@ptimization performed: dead branch removed
  /**/


  // Display text for amendur.dlg:0:20 if the conditions are all positive
  /*if(*//*@Optimization performed: constant pre-computation*//*FALSE*/ /*)*/
  /*return*/ /*FALSE;*/

  // Define Object Speaker as the object speaking in the amendur.dlg:0:20
  ObjectSpeaker_SE0 = OBJECT_SELF;

  // Define PC Speaker as the PC speaking in the amendur.dlg:0:20
  PCSpeaker_SE1 = GetPCSpeaker();

  // Define NPC Speaker as the NPC speaking in the amendur.dlg:0:20
  NPCSpeaker_SE2 = OBJECT_SELF;
  //@Optimization performed: unnessesary branch removed
  /**/


  // Main code body - checks conditions and executes actions

  // Sets up the multiCondition with selected BinaryDefinitions
  /*if(*/ /*TRUE*/ /*)*/
```

```
    /* This conversation node may be said.
     * All the conditions were true.
     */
    ConversationfilterWhenaconversationnodeisdisplayed_5_0 = TRUE;
  /**/

  /* Conversation node cannot be said. Some condition above
   * was false, so we fell through to here.
   */
  /*else*/
  /*return*/ /*FALSE;*/

//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  if( ConversationfilterWhenaconversationnodeisdisplayed_5_0 ) return TRUE;
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
  return FALSE;
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.11: scriptease6.nss before optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
void SE_Ac_ActionPlaceholder();
object SE_Df_DeadPlayer();
object SE_Df_ObjectKiller(object param_1);
int SE_Ev_ModuleOnPlayerDeath(object param_1);
void SE_Ac_ActionPlaceholder() {

}

object SE_Df_DeadPlayer() {
  return GetLastPlayerDied();
}

object SE_Df_ObjectKiller(object param_1) {
  return GetLastKiller();
}

int SE_Ev_ModuleOnPlayerDeath(object param_1) {
  return TRUE;
}

//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



void PlayerdeathDeadplayeraddactions_8();

void PlayerdeathDeadplayeraddactions_8() {

  // The following are all of the variables used in this situation

  object DeadPlayer_SE0;
  object Killer_SE1;


  // When a PC dies in Module
  if( ! SE_Ev_ModuleOnPlayerDeath(GetModule()) ) return;

  // Define Killer as the killer of Dead Player
  Killer_SE1 = SE_Df_ObjectKiller(DeadPlayer_SE0);

  // Define Dead Player as the player that died
  DeadPlayer_SE0 = SE_Df_DeadPlayer();

  // Define Killer as the killer of Dead Player
  Killer_SE1 = SE_Df_ObjectKiller(DeadPlayer_SE0);


  // Main code body - checks conditions and executes actions

  // Replace this action placeholder by one or more actions
```

```
  SE_Ac_ActionPlaceholder();


}

//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {
//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  PlayerdeathDeadplayeraddactions_8();
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.12: scriptease6.nss after optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
//@Optimization performed: function prototype declaration removed
//void SE_Ac_ActionPlaceholder();
//@Optimization performed: function prototype declaration removed
//object SE_Df_DeadPlayer();
//@Optimization performed: function prototype declaration removed
//object SE_Df_ObjectKiller(object param_1);
//@Optimization performed: function prototype declaration removed
//int SE_Ev_ModuleOnPlayerDeath(object param_1);
//@Optimization performed: function inlining, original declaration removed

//@Optimization performed: function inlining, original declaration removed

//@Optimization performed: function inlining, original declaration removed

//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



//@Optimization performed: function prototype declaration removed
//void PlayerdeathDeadplayeraddactions_8();
//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {

//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  //@Optimization performed: function inlining (PlayerdeathDeadplayeraddactions_8)


  // The following are all of the variables used in this situation

  object DeadPlayer_SE0;
  object Killer_SE1;
 //@Optimization performed: function inlining (SE_Ev_ModuleOnPlayerDeath)

int SE_Ev_ModuleOnPlayerDeath_0;
```

```
  SE_Ev_ModuleOnPlayerDeath_0 = TRUE;


  // When a PC dies in Module
  if( ! SE_Ev_ModuleOnPlayerDeath_0 ) return;
 //@Optimization performed: function inlining (SE_Df_ObjectKiller)

object SE_Df_ObjectKiller_1;
  SE_Df_ObjectKiller_1 = GetLastKiller();

  // Define Killer as the killer of Dead Player
  Killer_SE1 = SE_Df_ObjectKiller_1;
 //@Optimization performed: function inlining (SE_Df_DeadPlayer)

object SE_Df_DeadPlayer_2;
  SE_Df_DeadPlayer_2 = GetLastPlayerDied();

  // Define Dead Player as the player that died
  DeadPlayer_SE0 = SE_Df_DeadPlayer_2;
 //@Optimization performed: function inlining (SE_Df_ObjectKiller)

object SE_Df_ObjectKiller_3;
  SE_Df_ObjectKiller_3 = GetLastKiller();

  // Define Killer as the killer of Dead Player
  Killer_SE1 = SE_Df_ObjectKiller_3;



  // Main code body - checks conditions and executes actions

  // Replace this action placeholder by one or more actions
  //@Optimization performed: function inlining (SE_Ac_ActionPlaceholder)

//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.13: scriptease7.nss before optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



void ConversationactionopenstoreOpenastorewhenaconversationnodeisdisplayed_7();

void ConversationactionopenstoreOpenastorewhenaconversationnodeisdisplayed_7() {

  // The following are all of the variables used in this situation

  object ArtisanStore_SE3;
  object PCSpeaker_SE0;
  object ObjectSpeaker_SE1;
  object NPCSpeaker_SE2;


  // After anden.dlg:1:18 is reached
  if( ! TRUE ) return;

  // Define PC Speaker as the PC speaking in the anden.dlg:1:18
  PCSpeaker_SE0 = GetPCSpeaker();

  // Define Object Speaker as the object speaking in the anden.dlg:1:18
  ObjectSpeaker_SE1 = OBJECT_SELF;

  // Define NPC Speaker as the NPC speaking in the anden.dlg:1:18
  NPCSpeaker_SE2 = OBJECT_SELF;


  // Main code body - checks conditions and executes actions

  // Get the object with tag "ArtisanStore"
  ArtisanStore_SE3 = GetNearestObjectByTag("ArtisanStore");
  if (ArtisanStore_SE3 == OBJECT_INVALID )
    ArtisanStore_SE3 = GetObjectByTag("ArtisanStore");

  // Open Artisan Store for PC Speaker
  OpenStore(ArtisanStore_SE3, PCSpeaker_SE0, 0, 0);


}

//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {
//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
```

```
// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  ConversationactionopenstoreOpenastorewhenaconversationnodeisdisplayed_7();
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.14: scriptease7.nss after optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



//@Optimization performed: function prototype declaration removed
//void ConversationactionopenstoreOpenastorewhenaconversationnodeisdisplayed_7();
//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {

//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  //@Optimization performed: function inlining
  // (ConversationactionopenstoreOpenastorewhenaconversationnodeisdisplayed_7)


  // The following are all of the variables used in this situation

  object ArtisanStore_SE3;
  object PCSpeaker_SE0;
  object ObjectSpeaker_SE1;
  object NPCSpeaker_SE2;
  //@ptimization performed: dead branch removed
  /**/


  // After anden.dlg:1:18 is reached
  /*if(*//*@Optimization performed: constant pre-computation*//*FALSE*/ /*)*/
  /*return;*/

  // Define PC Speaker as the PC speaking in the anden.dlg:1:18
  PCSpeaker_SE0 = GetPCSpeaker();

  // Define Object Speaker as the object speaking in the anden.dlg:1:18
  ObjectSpeaker_SE1 = OBJECT_SELF;

  // Define NPC Speaker as the NPC speaking in the anden.dlg:1:18
  NPCSpeaker_SE2 = OBJECT_SELF;
```

```
  // Main code body - checks conditions and executes actions

  // Get the object with tag "ArtisanStore"
  ArtisanStore_SE3 = GetNearestObjectByTag("ArtisanStore");
  if (ArtisanStore_SE3 == OBJECT_INVALID )
    ArtisanStore_SE3 = GetObjectByTag("ArtisanStore");

  // Open Artisan Store for PC Speaker
  OpenStore(ArtisanStore_SE3, PCSpeaker_SE0, 0, 0);
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.15: scriptease8.nss before optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
void SE_Ac_ActionPlaceholder();
void SE_Ac_ActionPlaceholder() {

}

//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



void PlaceableunlockUnlockplaceableaddactions_10();

void PlaceableunlockUnlockplaceableaddactions_10() {

  // The following are all of the variables used in this situation

  object ThePlaceable_SE1;
  object Unlocker_SE0;


  // When Sundial is unlocked
  if( ! TRUE ) return;

  // Define Unlocker as the object that unlocked Sundial
  Unlocker_SE0 = GetLastUnlocked();

  // Define The Placeable as the object the event is fired on
  ThePlaceable_SE1 = OBJECT_SELF;


  // Main code body - checks conditions and executes actions

  // Replace this action placeholder by one or more actions
  SE_Ac_ActionPlaceholder();


}

//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {
//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
```

```
  PlaceableunlockUnlockplaceableaddactions_10();
//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```

Code A.16: scriptease8.nss after optimization

```
//@+SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE
#include "i_se_aux"
//@Optimization performed: function prototype declaration removed
//void SE_Ac_ActionPlaceholder();
//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(INCLUDE FILES) - DO NOT CHANGE THIS LINE

// Add your own include files here


//@+SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE



//@Optimization performed: function prototype declaration removed
//void PlaceableunlockUnlockplaceableaddactions_10();
//@Optimization performed: function inlining, original declaration removed


//@-SCRIPTEASE(FUNCTIONS) - DO NOT CHANGE THIS LINE

// Add your own functions here


//@+SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE
void main() {

//@-SCRIPTEASE(MAIN ENTRY POINT) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE
  //@Optimization performed: function inlining
  // (PlaceableunlockUnlockplaceableaddactions_10)


  // The following are all of the variables used in this situation

  object ThePlaceable_SE1;
  object Unlocker_SE0;
  //@ptimization performed: dead branch removed
  /**/


  // When Sundial is unlocked
  /*if(*//*@Optimization performed: constant pre-computation*//*FALSE*/ /*)*/
  /*return;*/

  // Define Unlocker as the object that unlocked Sundial
  Unlocker_SE0 = GetLastUnlocked();

  // Define The Placeable as the object the event is fired on
  ThePlaceable_SE1 = OBJECT_SELF;
```

```
  // Main code body - checks conditions and executes actions

  // Replace this action placeholder by one or more actions
  //@Optimization performed: function inlining (SE_Ac_ActionPlaceholder)

//@-SCRIPTEASE(MAIN CODE) - DO NOT CHANGE THIS LINE

// Add your own code here


//@+SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
}
//@-SCRIPTEASE(MAIN EXIT CODE) - DO NOT CHANGE THIS LINE
```