Bringing Verification to a Virtual World

William Moldenhauer, Dr. J.C. Browne, Dr. Calvin Lin

May 4th, 2007

Abstract

Second Life is a virtual world where avatars representing real entities interact through scripted transactions. Avatars may be anonymous so that relationships from the real world do not carry into Second Life, and their trustworthiness may not be apparent. Our goal is to apply software verification techniques to establish a basis for trust in Second Life within the realm of scripted transactions. Even though Second Life is a virtual world, it has an economy that provides very real financial opportunity as well as risk. Scripts can be used to help manage assets of financial value in Second Life, but can they be trusted to handle financial assets appropriately? This project aims to provide a solution to this problem by automatically generating models of Second Life scripts that can be verified through the use of traditional software verification tools. This verification capability serves as an initial step towards a certification system for Second Life, providing a framework for formally verifying that Second Life scripts correctly implement their desired behaviors and certifying scripts with varying levels of trust, based on the properties they can be formally shown to satisfy.

1. Introduction

[http://secondlife.com/whatis/] Second Life is a virtual world founded by Linden Lab but built by its residents. [Ondrejka 2003] The aim of Second Life is to act as a platform upon which users can create three-dimensional virtual spaces, and it is hoped that someday this will supplement the Internet in such a way that people visit three-dimensional places similarly to how they now visit two-dimensional pages. Currently everything is still hosted by Linden Lab, but the power to create content is readily available to all players of Second Life, allowing their collective creativity to shape the virtual world. Not only this, but players retain the intellectual property rights to the content that they create in Second Life, giving it value in the real world. [http://www.anshechung.com/include/press/press_release251106.html] Representing this value is a virtual currency known as the Linden dollar (L$), but there is an active market where this virtual currency can be exchanged for real money and vice-versa. This virtual economy creates very real financial opportunities, and it is not uncommon for over a million United States dollars to change hands within a twenty-four hour period within this virtual economy. In fact, one resident (Anshe Chung, known as Ailin Graef in the real world) has already become a real world millionaire off dealings in this virtual world. (see citation above)

But Second Life is not just a creative outlet and financial opportunity, it is also a social outlet and an intriguing medium for communication. Players are able to represent themselves as avatars whose appearance is limited primarily by their imagination. This freedom of expression allows avatars to represent their creators in the way in which they wish to be perceived, and appearance can have a real effect on how others view them. Players can express themselves with the descriptive capabilities of text chat, but text can be augmented by avatar animations, sounds, gestures, and other actions. Voice communication is also currently in the process of being integrated into Second Life, but has long been augmenting the experience through the use of third-party technology that can be run alongside Second Life, though not directly affiliated with Linden Lab. This level of expressive capability allows people to communicate in a way that is not really so different from the way we interact in the real world; it just happens virtually, and meaningful and important discussions do occur. Thinkers meet in a virtual setting to discuss real issues, and businesses hold virtual meetings to discuss their business, whether it operates in the real world, the virtual world, or both. While this form of interaction can parallel the way we interact in the real world, a key difference is that players do not see each other face to face, but rather avatar to avatar. The avatar is a mask that provides a level of anonymity not usually present in the real world. This mask helps some feel more free to express themselves, but empowers others to misbehave. In a world where we do not know who hides behind the avatar drawn on the computer screen, who can we trust?

There is a real need for trust in Second Life. Since there is real value in the virtual dealings there will be those who try to exploit this world for their own financial benefit or other ends. Players need some assurance that they are not being swindled in their virtual dealings. Some trust is achieved through the observance of longstanding behavior, online reputation, or association with trustworthy real-world entities, but these indicators do not provide any guarantees and can often be misleading.

This need for trust is largely a social problem, but there is a smaller problem that is easier to confront. Many dealings within Second Life occur through special scripting that is written by players. This scripting can manage the sale or rental of intellectual property, conduct games, give prizes, give users automated access to a virtual bank, and so much more. Players who interact with these scripts have expectations about how they should behave, but currently there is no framework to formally establish properties of these scripts. This smaller problem is open to a technical solution, and if these scripted exchanges and virtual interactions which in fact carry real value could be formally verified against sets of properties, then perhaps some real trust could be established. This project takes a step in this direction.

The following sections will give background information on Second Life, Linden Scripting Language (LSL) [http://wiki.secondlife.com/wiki/LSL_Portal], Promela [http://spinroot.com/spin/Man/promela.html], and property specification in Linear Temporal Logic (LTL) as it applies to SPIN [http://spinroot.com/spin/Man/ltl.html]. Then the translation of Second Life scripts to models will be discussed as a step towards establishing a framework to formally establish trust. But there are some limitations, so guidelines for writing scripts to be verified will be presented in the following section. A case study is then included to exhibit the basic functionality of this system, followed by related work, lessons learned, future work, and lastly conclusions.

2. Background

2.1 Second Life

[http://en.wikipedia.org/wiki/Resident_(Second_Life)] In Second Life, users inhabit virtual land and so are also known as "residents". Residents connect to Second Life servers, collectively called a grid, through a protocol defined by Linden Lab, usually using the client provided by Linden Lab which is now open-source. The client program must first verify the resident's virtual identity with the login server, and then if approved is let into a "simulator", also known as a "region" or "sim" for short. Each region simulates an area of virtual land and represents some allocated amount of resources from the grid. The Second Life world consists of a few mainland continents, which are large land masses consisting of many simulators that were initially created by Linden Labs and sold to various residents. Residents can also submit orders to purchase private islands which exist off the coasts of the mainland continents, and these have become far more popular since they give their owners more complete control. Specifically, for another simulator to border the island, express written permission must be given by the owner of the island in question, giving the private island owner assurances that undesirable neighbors will not move in next door, which has been a common complaint of many residents who own land on the mainland continents. Private island owners are also given more powerful management tools to control their simulator, but the simulators, both on the mainland and private islands, are still run and operate on Linden Lab servers. But even though everything is currently hosted by Linden Lab, Second life is a platform for user-created content.

Residents of Second Life control "avatars", which are virtual representations of themselves. The basic avatar has a humanoid shape, which can be customized by adjusting various parameters such as body thickness and height. The customization potential is vast, allowing residents to represent themselves in new and interesting ways that do not necessarily correspond to their actual real world characteristics. User-created objects can even be attached to avatars to expand the possibilities, allowing users to add appendages or create entirely new species.

Any resident can create objects in Second Life. Objects are built from primitive shapes which are called “prims” for short. The process through which prims are joined together to form objects is known as “linking”, and objects are also sometimes referred to as “linksets”. The members of a linkset have an ordering which is determined by the order in which the individual prims are selected to be linked by the user. The last prim selected will become the “root” of the linkset. Objects and other forms of created content can be collectively referred to as “assets”. Inventory refers to a collection of assets. All residents have an inventory, and objects can have an inventory as well. The root prim contains the main inventory of the object, but the other prims can hold inventory as well. Inventory can consist of a variety of types of virtual assets created by residents, but the main type of inventory we are interested in here is scripts.

[http://secondlife.com/knowledgebase/article.php?id=166] Scripts allow an object to become interactive. Without scripts an object does not do anything on its own, but a script allows the object to react to various types of events. This allows scripts to interactively perform various functions from technical functions (like mathematic calculations or managing financial assets) to expressive functions (like creating firework-like particle effects and helping avatars dance). Some functions will also trigger other events in the script or even send messages to other scripts. Scripts can even cause objects to speak. Text chat in Second Life operates on various “channels”, the default being channel 0 which is automatically heard by avatars and is the channel on which avatars speak, unless they specify otherwise. Scripts can chat on channel 0 to speak to avatars that are nearby or can communicate silently with other objects that are listening for messages on a non-zero channel.

2.2 Linden Scripting Language

The Linden Scripting Language (LSL) is an event-driven state-machine language. The basic syntax and operators are reminiscent of popular languages like C++ or Java, but the structure is fundamentally different.

Each script consists of global variables/functions and a list of states. Each state defines event handlers that define the behavior of the script while in that state. An event handler can perform arithmetic operations, call functions, trigger state transitions, and make use of familiar conditional statements and control flow structures. The environment, which consists of user input and server interaction, generates events that trigger the event handlers defined in the script based on the current state the script is in.

Below is a simple example of a script:

```
default
{
 state_entry()
 {
  llSay(0, "Hello World!");
 }
}
```

This example includes only one state, which is the default state that must be implemented. It is also necessary that there is one event handler within the state, and here the *state_entry* event was defined to instruct the object to speak the phrase "Hello World" on channel 0. So this script will cause the object to speak and then sit silently, taking no further actions since the *state_entry* event will only be triggered when the state is entered. Scripts can also include more than one state, as in the next example.

```
default
{
 state_entry()
 {
  llSay(0, "Off!");
 }
 touch_start(integer num_detected)
 {
  state on;
 }
}

state on
{
 state_entry()
 {
  llSay(0, "On!");
 }
 touch_start(integer num_detected)
 {
  state default;
 }
}
```

This example implements a simple on/off switch that announces whether it is on or off. The *touch_start* event is triggered when a user clicks on the object, allowing a user to click to transition the script from one state to the other. More complex examples can have even more states, each of which responds to different sets of event handlers.

2.3 Model Checking and Promela

Model checking is a formal method of verification. The goal is to formally prove that a system implements a specified behavior. Model checking exhaustively searches all reachable states of a system to ensure that the required behavior is followed. SPIN is an example of a model-checker and is used in this project. Promela is a modeling language designed to model concurrent processes for verification. Promela is designed well for verification and can be directly model checked by the SPIN model checker.

In Promela, the behavior of a process is defined by a *proctype* (derived from process type). A *proctype* may be declared as *active* which indicates that it is to be running initially when the model is executed. If no *proctype* is declared as active, then the model must define an *init* process which can then run processes defined by *proctype* definitions.

The simple logical and arithmetic statements are similar to other languages such as C, but one fundamental concept of Promela that may seem foreign to traditional programmers is the notion of *executability*, which applies to each statement of the language. Traditional procedural programming languages are generally defined to execute a sequence of instructions in a non-blocking manner, but in Promela statements naturally block execution until (and if) they become true. So in modeling procedural steps of a script, care must be taken to ensure that standard statements that would execute in the script do not block in the model.

Promela supports only simple data-types, including numeric types such as *bit*, *byte*, and *int*. A special data-type is *chan*, a channel, which is a means of passing messages between processes. Processes can send values in a channel by specifying the channel variable followed by an exclamation point and then the value to send. For example, "ch!5" would send the value 5 into the channel represented by the variable ch. Similarly, processes can receive values from a channel using a question mark. For example, "ch?x" would receive a value from the channel ch and store it in x. The expression following the question mark can also be a constant, in which case the statement is executable if that value is received from the channel. For example, "ch?5" would not be executable unless 5 is the value that can be received from the channel.

Control-flow structures seem familiar to traditional procedural languages but are actually a little different. Below is an example of a simple *if* statement.

```
if
:: x < 10 -> x++;
:: else -> skip;
fi
```

This will increment the variable 'x' if it is less than 10; otherwise it will skip, which is an instruction that performs no operation. Note that if the else case had not been included, this conditional would have blocked execution until x becomes less than 10. The basic looping structure, *do*, is shown below.

```
do
:: x < 10 -> x++;
:: x < 5 -> x--;
:: else -> break;
od
```

Similarly to the previous example, this will increment the variable 'x' if it is less than 10. But now notice there is another case and further that the cases overlap. Execution blocks until at least one case becomes true and then if more than once case is true, non-deterministically selects a case to execute. In this example, the *else* case is again included to ensure that there is always an executable case. Here, the *else* case will break, preventing the loop from being repeated, otherwise the loop will start over and select a new case after executing. Thus this example will either increment or decrement 'x' while it is less than 5, increment x when it is greater than or equal to 5 (but less than 10), and will break when x becomes equal to 10. Note that termination is not guaranteed however, it is always possible to continuously increment and decrement 'x' to keep it in a range that is less than 5.

Promela also supports labels and goto statements. Labels are denoted with a name followed by a colon, ":", followed by the corresponding code. For example, "foo: skip" is an example of a label named "foo" which then executes a skip statement. It is then possible to say "goto foo" in order to jump to the "foo" label.

2.4 Linear Temporal Logic and Spin Property Specification

Properties are a formal specification of the required behaviors of a system. Properties are formally specified in a temporal logic, such as linear temporal logic (LTL). LTL incorporates the standard logic operators such as *and* (&&), *or* (||), *not* (!), *implication* (->), and *equivalence* (<->) along with the temporal operators: *always* ([]), *eventually* (<>), *until* (U), and *release* (V). These operators are explained below.

Operator	**Syntax**	**True if and only if ...**
and	p && q	p and q are both true
or	p || q	at least one of p or q is true
not	!p	p is false
implication	p -> q	!p || q
equivalence	p <-> q	(p && q) || (!p && !q)
always	[] p	p is always true
until	p U q	q eventually becomes true and p is true until then (p may still be true after)

Operator	Syntax	True if and only if ...
release	p V q	q is true until after the first position in which p is true (p is not required to become true, p is said to "release" q from the requirement of being true)

Spin can recognize properties specified in LTL as claims about system behavior. These claims can include most Promela language constructs, primarily excluding those which have side effects such as assignment. Spin takes the strategy of verifying properties which should never occur, so if the behavior described is a desirable one, it should first be negated. The LTL is translated into an executable process that verifies the behavior described, represented by a *never* claim. It is sufficient to understand LTL, for more information on Promela *never* claims, refer to [http://spinroot.com/spin/Man/promela.html]. A simple LTL example from the site with the corresponding *never* claim is included below:

```
never { /* <>[]p */
        do
        :: true /* after an arbitrarily long prefix */
        :: p -> break             /* p becomes true */
        od;
accept: do
        :: p      /* and remains true forever after */
        od
}
```

3. Problem Solution

In order to create a framework to formally establish properties of Second Life scripts, the Promela modeling language will be used to model Second Life scripts. This decision was made because formal properties about a Promela model can be model-checked by the popular SPIN tool, which is under ongoing development and innovation.

The Second Life client has been open-sourced and includes C++ source code that implements a compiler for the Linden Scripting Language (LSL). This project has made use of the tokenizing and parsing elements of this compiler to aid in translation. The tokenizer is defined in the format used widely by the popularized *lex* program, which generates lexical analyzers. The parser for the tokens is defined in the format used by the popular complement to *lex*, which is *yacc*, used for generating parsers. It is not necessary to know the specifics of these programs, other than that they split a script into tokens and parse the tokens into a tree structure which represents the script. The compiler would then compile this code to the lower-level code that is executed on Second Life servers. For this project, the tree is instead used to translate the script into a Promela model for verification.

From a high-level view, the abstract syntax tree closely resembles the structure of the language. Each node represents a concept of the language and includes pointers to the concepts that together form the concept of the node. Some of these concepts appear in the language in a list-like fashion, and the nodes for these concepts act as a linked list, each node having a pointer to the next. Each node also contains a method that is used to compile the script in successive passes by doing some work and then recursively calling the method on the nodes to which it points. This method is not of particular interest for this project, but since the members of all the nodes are public, the tree can be traversed in a similar fashion from a separate class to generate a model.

A translator has been created as part of this project to automatically produce Promela models from LSL scripts. The translator makes use of the tokenizer and parser from the Second Life LSL compiler and implements new functions in C++ to convert the tree structure generated by the parser into the textual representation of Promela. The general structure of the translator makes use of a hierarchy of recursive functions that very closely correspond to the elements of the tree structure. These functions generate the Promela equivalent of the LSL concept, which is in many cases straight-forward. Special care is taken with control-flow statements to include else clauses so that they are always executable. Since Promela does not support scopes beyond global and local, scopes represented in LSL must be merged together. Variables are renamed with a prefix to denote their scope if conflicts occur.

An LSL script is translated into a single proctype definition. LSL states correspond to labels in the Promela model. The Promela code for the LSL state_entry event (if defined) is included after the label. The model then includes a *do* loop that receives an event from the event queue, with cases for each event handler which will execute the corresponding code to handle the events. Parameters to the events are non-deterministically generated. State transitions are then modeled as goto statements. If there is an LSL state_exit event, the corresponding Promela code is inlined before the state transition. A template of a script *proctype* is included below, some parts are explained in subsequent paragraphs:

```
proctype script(chan event_queue; chan event_request)
{
        /* variable declarations */
        state_default:
        /* state_entry */
        event_request!0;
        state_default_loop:
        do
        :: event_queue?/* event number */ -> /* translated event handler */ event_request!0;
        ...
        :: event_queue?/* event number */ -> /* translated event handler */ event_request!0;
        od;

        state_two:
        /*state_entry*/
        event_request!0;
        state_two_loop:
        do
        :: event_queue?/* event number */ -> /* translated event handler */ event_request!0;
        ...
        :: event_queue?/* event number */ -> /* translated event handler */ event_request!0;
        od;


        state_closed:
        /*state_entry*/
        event_request!255;
}
```

In order to allow for model checking, the model must be "closed", meaning there are no inputs. So an environment process is generated to close the model. The environment acts as the active *proctype* and runs the process corresponding to the LSL script. It is then responsible for generating events by sending them through the event queue. The environment will non-deterministically generate an event for which the script model has an event handler. A template for the environment *proctype* is included below, some parts to be explained later:

```
active proctype environment()
{
        chan event_queue = [0] of { byte };
        chan event_request = [0] of { byte };
        byte e;
        byte timer=0;

        run script(event_queue,event_request);

        generate_event:
        event_request?e;
        if
        :: e == 0 ->
                if
                ::timer>1 -> timer--;
                ::else -> skip;
                fi;
                if
                ::timer==1 ->
                        do
                        :: timer< /* maximum number of events within a timed delay */ -> timer++
                        :: break
                        od;
                        event_queue!12;
                ::else -> /* non-deterministically generate events */
                fi;
        :: e == 12 ->
                timer=1;
                do
                :: timer< /* maximum number of events within a timed delay */ -> timer++
                :: break
                od;
        :: e == 255 -> goto terminate;
        :: else -> event_queue!e;
        fi;
        goto generate_event;
        terminate:
        skip;
}
```

The model must also be guaranteed to terminate in order to be model checked. So an additional channel is used to allow the script process to request events by sending a zero value. The LSL script must eventually reach a state where it no longer responds to any events at which point it terminates. For such a state, the translator will not include an event handling loop but instead will signal to the environment that it should terminate by sending the maximum value on the event request channel. The script process will then terminate, and the environment process will terminate upon receiving the maximum value.

Functions cannot be directly represented in Promela, so user-defined functions are translated and inlined in the Promela model. Built-in LSL functions are abstracted from the Promela model. Some of these functions will trigger other events and this is achieved through sending a non-zero value on the event request channel. Upon receiving a non-zero value, the environment will generate the event specified by that value. These special events are not generated by the environment otherwise, as it is assumed that these events should only be triggered in response to these function calls. A special case of this is the llSetTimerEvent function which accepts a parameter specifying a delay for a periodic timer, triggering the timer event after each delay. In this case the script process will request a timer event through the event request channel and the environment will non-deterministically pick a number of events to occur within the delay. This value is held in a variable within the environment and is then decremented each time an event is generated. This variable is initially zero, but when the timer is active will hold a non-zero value and upon reaching one will non-deterministically pick a number of events to occur within the next delay and then triggers the timer event.

Special options included to assist in verification are the modeling of function arguments and of the function call itself. This is achieved through a sequence of variables progressively named “arg0”, “arg1”, etc. up to the maximum number of arguments required to be modeled. These variables are assigned the values of parameters passed to a function. Argument modeling can be enabled or disabled on a per function basis. Ideally all arguments would be modeled and a slicing algorithm would be used to eliminate those which are not relevant. The function call itself is represented by toggling a bit variable named for the function to true, followed by an abstraction of the function, and then toggling the bit variable back to false so that it can be used again. The only function abstractions implemented at this time are those which will trigger events, all others are abstracted as a skip statement. These options allow claims to be made about arguments and function calls.

Special care is also given to assignment statements. To allow claims to be made about when assignments occur, a bit toggling strategy is again used. A bit variable named for the variable in question with an additional suffix such as "_assign" is toggled to true before the assignment and toggled to false just after the assignment. This is provided as a convenient way to make claims about assignment of variables.

In addition to modeling the script itself, formal properties about the behavior of the script need to be specified. Ideally, properties would be specified as LSL annotations allowing LSL itself to be augmented with temporal operators to specify properties, which would be translated along with the LSL itself, but this is left for future work due to time constraints. For now, properties must be specified in Promela claims.

4. Guidelines for Writing Verifiable Scripts

Due to current limitations in software verification technology, it may not be possible to verify arbitrary scripts written in Linden Scripting Language. This is because LSL offers a variety of complex data types that cannot be directly modeled. Complexity also contributes to the problem of state-space explosion, where the possible execution paths of a program grow too quickly and become too many to verify with current technology in a practical time-frame. This is particularly true in the case of non-termination, so it is necessary to ensure that all execution paths terminate by reaching a state which no longer handles further events. Taking this in mind, it is best to keep scripts as simple as possible and resort to complex data types only where necessary.

Another guideline to help in verification is to incorporate state variables into your program. These variables should be named descriptively, in a unique way, so that they do not clash when scopes are merged. This will allow properties to be specified about these variables without including a scope prefix.

5. Case Study

To exhibit the usefulness of this solution, an example of its use is necessary. The examples most fitting for this seem to be those where the most risk is involved and thus the greatest need for trust: those which handle the exchange of (virtual) money and intellectual property. Players who pay a scripted object, grant a scripted object permission to debit their account, or entrust a script to manage the rental or sale of their intellectual property wish to know that it handles their assets appropriately and behaves as expected. Properties can be specified and verified to ensure that the expected behaviors are followed.

Ideally, requirement specifications would drive the software design process toward verification. There would then be a clear set of properties to prove about a script, and it would be designed with verification in mind. But this is not commonly the case with LSL scripting. Usually there would be multiple people involved in this process and those who specify the requirements are often different from those who design the software and perform the verification, but the following example has been produced entirely by the author for illustrative purposes.

The case study that has been chosen is a script to manage the auction of in-world objects. The setup in-world consists of a scripted object which has in its inventory an item to be auctioned. The owner will initiate the auction and then other avatars may bid by paying the object Linden dollars. The object will refund bids which are out-bid until a fair timer decides that the auction has ended at which time the prize will be awarded to the highest bidder, who is not refunded his or her high bid. The LSL for this study is included below:

```
string prize;
string high_bidder_name;
key high_bidder;
integer high_bid;

default
{
 state_entry()
 {
    llSetText("Initializing", <1,1,1>, 1.0);
    prize=llGetInventoryName(INVENTORY_OBJECT, 0);
    llSetText(prize, <1,1,1>, 1.0);
    llRequestPermissions(llGetOwner(), PERMISSION_DEBIT);
 }

 run_time_permissions(integer perm)
 {
    state auction;
 }
}

state auction
{
 state_entry()
 {
    high_bidder=NULL_KEY;
    high_bid=0;
    llSetText(prize+" \n Bidding Open! \n 0", <1,1,1>, 1.0);
    llSetTimerEvent(300);
 }

 money(key id, integer amount)
 {
    if(amount > high_bid)
    {
       if(high_bidder != NULL_KEY)
       {
          llOwnerSay("Refund:"+(string)high_bidder+","+(string)high_bid);
          llGiveMoney(high_bidder, high_bid);
       }
       high_bidder=id;
       high_bidder_name=llKey2Name(high_bidder);
       high_bid=amount;
             llSetText(prize+" \n "+high_bidder_name+" \n "+(string)high_bid,
<1,1,1>, 1.0);
    }
    else
    {
       llOwnerSay("Refund:"+(string)id+","+(string)amount);
       llGiveMoney(id, amount);
          llInstantMessage(id, "You did not offer enough to outbid the highest
bidder");
    }
 }

 timer()
 {
    state closed;
 }
}

state closed
{
 state_entry()
 {
    if(high_bidder != NULL_KEY)
    {
         llSetText(prize+" \n Bidding Closed! \n "+high_bidder_name+" wins! \n
"+(string)high_bid, <1,1,1>, 1.0);
       llOwnerSay("Award:"+(string)high_bidder+","+prize);
       llGiveInventory(high_bidder, prize);
    }
    else
    {
       llSetText(prize+" \n Bidding Closed.", <1,1,1>, 1.0);
    }
 }
}
```

The auction is initialized in the *default* state, but the auction itself occurs within the *auction* event, which sets the timer and accepts bids through the *money* event, keeping track of the high bidder and high bid. When the timer event is triggered, the script transitions to the closed state where the prize is awarded if there was a high bidder. This script has been translated to a Promela model which is included as an appendix.

There are several behavioral expectations in this example, and the potential for danger is high since users commit funds as bids when they pay the object. The required behaviors can be stated in English and then cast as LTL claims to provide an executable specification to SPIN.

The first property that is of interest is termination. In this example, if the model does not terminate, then no other properties may be verified. The environment process includes a "terminate" label that is reached when the script process signals that it is terminating by sending the maximum value in the event request channel.

So the termination properly can be stated simply by the temporal claim: "Eventually the environment reaches the terminate state". This is cast into LTL as "<> environment@terminate" from which an executable *never* claim is produced by SPIN and used in verification. The correctness properties verified are summarized in the table below:

English	**Desired LTL behaviors**	***never***

High bid is never replaced by an equal or lower bid	[] !((auction:amount <= auction:high_bid) && (auction:high_bid_assign)	never {
		T0_init:
		    if
		    :: ((auction:amount <= auction:high_bid)
		      && (auction:high_bid_assign)) -> goto accept_all
		    :: (1) -> goto T0_init
		    fi;
		accept_all:
		    skip
		}

No bidder becomes high bidder unless (s)he outbid the current high bidder	[] !((auction:amount <= auction:high_bid) && (auction:high_bidder_assign)	never {
		T0_init:
		    if
		    :: (auction:high_bidder_assign &&
		      (auction:high_bidder!=0) &&
		      (auction:amount<=auction:high_bid)) ->
		      goto accept_all
		    :: (1) -> goto T0_init
		    fi;
		accept_all:
		    skip
		}

If there is a high bidder, then the prize is eventually awarded to the high bidder.	(<>auction:high_bidder) ->
	(<>
	((auction:arg0==auction:high_bidder) &&
	(auction:arg1 == auction:prize) &&
	auction:llGiveInventory))
		never {
		T0_init:
		    if
		    :: (! ((auction:arg0==auction:high_bidder) &&
		          (auction:arg1==auction:prize) &&
		           (auction:llGiveInventory)) &&
		      (auction:high_bidder)) -> goto accept_S4
		    :: (! ((auction:arg0==auction:high_bidder) &&
		          (auction:arg1==auction:prize) &&
		           (auction:llGiveInventory))) -> goto T0_init
		    fi;
		accept_S4:
		    if
		    :: (! ((auction:arg0==auction:high_bidder) &&
		          (auction:arg1==auction:prize) &&
		           (auction:llGiveInventory))) -> goto accept_S4
		    fi;
		}

English	Desired LTL behaviors	*never*

Prize is not awarded to anyone else	[] !((auction:arg0 != auction:high_bidder) && (auction:arg1 == auction:prize) && auction:llGiveInventory)	never {
		T0_init:
		    if
		    :: (! ((auction:arg0==auction:high_bidder)) &&
		      (auction:llGiveInventory==1)) -> goto accept_all
		    :: (1) -> goto T0_init
		    fi;
		accept_all:
		    skip
		}

The current high bidder is not refunded unless outbid	[] !((auction:llGiveMoney==1 &&
	        auction:arg0==auction:high_bidder)
	&& (auction:id!=auction:high_bidder &&
	        auction:amount<=auction:high_bid))
		never {
		T0_init:
		    if
		    :: ((auction:llGiveMoney==1 &&
		        auction:arg0==auction:high_bidder) &&
		      (auction:id!=auction:high_bidder &&
		       auction:amount<=auction:high_bid)) ->
		         goto accept_all
		    :: (1) -> goto T0_init
		    fi;
		accept_all:
		    skip
		}

A bidder who just bid is not refunded unless (s)he underbid	[] !((auction:llGiveMoney==1 &&
	        auction:arg0==auction:id) &&
	(auction:high_bidder!=auction:id &&
	        auction:amount>auction:high_bid))
		never {
		T0_init:
		    if
		    :: ((auction:llGiveMoney==1 &&
		        auction:arg0==auction:id) &&
		      (auction:high_bidder!=auction:id &&
		        auction:amount>auction:high_bid)) ->
		         goto accept_all
		    :: (1) -> goto T0_init
		    fi;
		accept_all:
		    skip
		}

Each of these cases was tested with the Promela model as well as a similar model that violates the property. Variables which are generated non-deterministically were constrained in testing to produce faster results. Unconstrained values dramatically increase the complexity of searching the model on all execution paths, and even a single 32-bit integer is stated to be well beyond the scope of a state-based model checker on [http://spinroot.com/spin/Man/float.html]. And for this case study, constrained values still produce all the interesting cases for the model. The primary sources of complexity in the model are the ranges in which various values are non-deterministically generated, namely the number of events that will occur in a timer duration as designed in the environment and the range of keys and monetary amounts that will be given to the money event within the auction itself. Tests were performed on a Pentium 4 Xeon processor with a total of 8 cores and 32 GB of RAM memory. Constrained cases can complete in a matter of minutes, but as these values reach even byte ranges the verification can begin to take days. Extremely constrained cases (such as up to 10 bids, 5 unique bidders, and bids of up to L$10) can complete in about a minute yet use nearly 200 MB of memory. Less constrained cases (such as up to 50 bids, 10 unique bidders, and bids up to L$100) can take several hours and several gigabytes of memory – depending on the specific property in question and the specific parameters used that affect initialization. Full byte ranges were found to run for days without terminating and press at the limits of memory, though there may be ways to optimize verification parameters to improve this. Ideally, these would be full integer ranges as that is what the semantics of LSL supports, but this is beyond the scope of SPIN.

6. Related Work

Model checking is a broad research field with many different methods, tools and approaches. To our knowledge this is the first application of formal verification to Second Life scripting. But a known approach to model checking has been used, utilizing a single method and tool. This section primarily focuses on research related to our approach.

There are two approaches to model checking. One is to specify the system to be model checked directly in a model checkable language. The other is to specify the system to be model checked in a conventional programming language and translate this program into a model checkable language. A survey of approaches and a summary of research on the translation based approach can be found in [Havelund and Visser 2002]. The approach we applied is the translation based approach.

The advantages of the translation based approach are: (i) the developer of the software system gets to write the software in a familiar representation, (ii) the possibility of transcription errors in recoding a verified model to a software representation is avoided and (iii) reuse is made of the enormous effort which goes into building and maintaining a powerful model checking system.

Most previous work using the translation based approach has been on C [Holtzman 2000, Ball and

Rajamani 2000], Java [Havelund and Pressburger 1998, Demartini, Iosif, and Sisto 1999, Corbett, et.al. 2000] or xUML [Xie and Browne 2002a, Xie and Browne 2002b]. Translation of software is most often done by translation to Promela/Spin [Holtzman 2003] because the semantics of the Promela input language for SPIN is more similar to conventional programming languages than is the case for other model checkable languages although Xie and Browne translate xUML to SR, the language of the CoSpan model checker.

We translate LSL to Promela, apply the Spin model checker, and formulate properties to be verified in linear temporal logic (LTL) (See for example, Chapter 3 of [Clarke, Grumberg and Peled 2003] for an introduction to temporal logics.)

Readily accessible general references on model checking include [Clarke, Grumberg and Peled 2003, Berard, et.al. 1998, Holtzman 2003, Kurshan 1994]

7. Lessons Learned

If I were to attack this problem again, I would look to more immediately move to the verification phase by first manually translating a script and then use what is learned to improve the translator. The approach taken was to first "finish" the translator, but I ended up tweaking the translator afterward in response to my experiences with verifying properties. This could take on a quick cycle where the translator is continually tweaked to make verification easier and the experiences in verification feed back into the tweaking of the translator. I feel this would be an effective strategy since I did not know enough about model checking with SPIN at the start, and I feel I did not get into this cycle soon enough. I would also consider first performing quick manual translations of simple scripts into a variety of targets for comparison.

8. Future Work

There is still more room to apply more advanced model-checking techniques as well as other forms of verification such as automated theorem-proving. This could lead to a more extensive, comparative study that displays the effectiveness of various solutions. A particular interest would be techniques which allow for the verification of scripts which do not necessarily terminate, since many scripts in Second Life are designed to execute continuously in the virtual world.

Property specification is difficult at this time, and this is a common problem. Ideally, formal properties could be specified in LSL syntax, augmented with temporal operators. Such properties would then be translated to the necessary form for the verification tools used. It would then be easier for those who know LSL to formulate properties about their code. This method can be generalized as a unified approach to verification and validation in which programmers specify properties in their own language, augmented with temporal operators.

An original goal which was abandoned for now due to time constraints was to also translate LSL-style syntax into a traditional programming language, preserving elements which are not recognized as valid LSL. This allows statements from the target language to be embedded into LSL-style code which is translated into the target language, producing a traditional application in a non-traditional manner. The state machine structure is convenient for verification and also seems like it could be convenient for various types of applications which are event-driven, such as graphical user interfaces. There are some software tools available for converting state maps into code which must be augmented with implementation code in order to produce the finished application. Examples of such tools include SMC and SmartState. I believe an LSL-style syntax would potentially be more convenient in that a programmer would not need to think separately to define a state map and then alongside that define implementation code to drive the state machine, but instead the programmer would just define the state machine in LSL-style syntax, allowing the programmer a single focus that is in my opinion more easily conceptualized, though this may certainly be a matter of preference.

Another worthwhile endeavor could be to establish a library of formally verified components. Users would then be able to use these components with their scripts and know exactly how the components can be expected to behave. Taking this a step further, automatic programming could be applied to allow a tool to locate the components necessary to create a script with a desired behavior. This would make programming Second Life scripts easier and help prevent errors by providing users with formally verified components.

I feel Second Life also has great potential for interesting work related to other fields of computer science. The compiler that is used to compile Second Life scripts does not perform optimizations. Thus, a beneficial work would be to implement optimizations for improving LSL code. Standard optimizations and techniques could be applied, but it may be interesting to consider if there are new approaches or classes of optimizations that can be applied, since the state-machine design of LSL is structurally different from traditional/popular programming languages.

The anonymity of avatars also creates a great opportunity for a classic challenge of artificial intelligence: to fool a human into thinking a computer program is human. The Second Life client and protocol are now open to the public, so it is possible to program automated bots which connect to Second Life as avatars. The level of anonymity here as well as the lack of sensual perception makes this challenge more feasible, but the bot would still need to chat and move around in the virtual world in a believable manner.

Second Life itself is also a great example of distributed computing and there are many associated challenges here. While Linden Lab does have long-term plans to open-source the server software, it is still proprietary at this time, so potential in this area is limited. However, there are open-source efforts to produce similar server software compatible with Second Life clients and such efforts may provide challenges in distributed computing. These can be seen at [http://openmetaverse.org/wiki/OpenSim]. Virtualization would also be an area of interest here, as currently each Second Life simulator represents a piece of physical hardware and thus a fixed allocation of resources, for more information on this topic see: [http://en.wikipedia.org/wiki/Virtualization].

9. Conclusions

The work presented here is an initial step toward establishing a framework for formally establishing trust in Second Life through the use of software verification techniques. There is still work to be done, but this study provides the capability to formally establish properties of a subset of Second Life scripts, namely those which are guaranteed to terminate and whose correctness properties do not rely upon complex data structures.

10. Appendix: Promela Model for Auction Case Study

```
active proctype environment()
{
        chan event_queue = [0] of { byte };
        chan event_request = [0] of { byte };
        byte e;
        byte timer=0;

        run auction(event_queue,event_request);

        generate_event:
        event_request?e;
        if
        :: e == 0 ->
                if
                ::timer>1 -> timer--;
                ::else -> skip;
                fi;
                if
                ::timer==1 ->
                        do
                        :: timer<T -> timer++ /* inserted note: T=maximum value for events between timer delay */
                        :: break
                        od;
                        event_queue!12;
                ::else -> event_queue!18;
                fi;
        :: e == 12 ->
                timer=1;
                do
                :: timer<T -> timer++ /* inserted note: T=maximum value for events between timer delay */
                :: break
                od;
        :: e == 255 -> goto terminate;
        :: else -> event_queue!e;
        fi;
        goto generate_event;
        terminate:
        skip;
}

proctype auction(chan event_queue; chan event_request)
{
        byte arg0;
        byte arg1;
        bit llSetText;
        bit llGetInventoryName;
        bit llRequestPermissions;
        bit llSetTimerEvent;
        bit llOwnerSay;
        bit llGiveMoney;
        bit llKey2Name;
        bit llInstantMessage;
        bit llGiveInventory;
```

```
byte prize;
bit prize_assign;
byte high_bidder_name;
bit high_bidder_name_assign;
byte high_bidder;
bit high_bidder_assign;
byte high_bid;
bit high_bid_assign;

byte perm;
bit perm_assign;
byte amount;
bit amount_assign;
byte id;
bit id_assign;

state_default:
llSetText=1;
skip;
llSetText=0;
llGetInventoryName=1;
skip;
llGetInventoryName=0;
prize_assign=1;
prize=0;
prize_assign=0;
llSetText=1;
skip;
llSetText=0;
llRequestPermissions=1;
event_request!24;
llRequestPermissions=0;
state_default_loop:
do
:: event_queue?24 ->
        perm=255;
        goto state_auction;
:: event_queue?18 -> skip;
:: event_queue?12 -> skip;
od;

state_auction:
high_bidder_assign=1;
high_bidder=0;
high_bidder_assign=0;
high_bid_assign=1;
high_bid=0;
high_bid_assign=0;
llSetText=1;
skip;
llSetText=0;
llSetTimerEvent=1;
skip;
llSetTimerEvent=0;
event_request!12;
event_request!0;
state_auction_loop:
do
:: event_queue?18 ->
        id=1;
```

```
                do
                :: id<K -> id++; /* inserted note: K=number of unique bidders (keys) */
                :: break
                od;
                amount=0;
                do
                :: amount<L -> amount++ /* inserted note: L=range for monetary amounts in L$ */
                :: break
                od;
                if
                ::amount>high_bid ->
                        if
                        ::high_bidder!=0 ->
                                llOwnerSay=1;
                                skip;
                                llOwnerSay=0;
                                arg0=high_bidder;
                                arg1=high_bid;
                                llGiveMoney=1;
                                skip;
                                llGiveMoney=0;
                        ::else -> skip;
                        fi;
                        high_bidder_assign=1;
                        high_bidder=id;
                        high_bidder_assign=0;
                        llKey2Name=1;
                        skip;
                        llKey2Name=0;
                        high_bidder_name_assign=1;
                        high_bidder_name=0;
                        high_bidder_name_assign=0;
                        high_bid_assign=1; skip;
                        high_bid=amount;
                        high_bid_assign=0;
                        llSetText=1;
                        skip;
                        llSetText=0;
                ::else ->
                        llOwnerSay=1;
                        skip;
                        llOwnerSay=0;
                        arg0=id;
                        arg1=amount;
                        llGiveMoney=1;
                        skip;
                        llGiveMoney=0;
                        llInstantMessage=1;
                        skip;
                        llInstantMessage=0;
                fi;
                event_request!0;
        :: event_queue?12 ->
                goto state_closed;
        :: event_queue?24 -> skip;
        od;

        state_closed:
        if
        ::high_bidder!=0 ->
```

```
                llSetText=1;
                skip;
                llSetText=0;
                llOwnerSay=1;
                skip;
                llOwnerSay=0;
                arg0=high_bidder;
                arg1=prize;
                llGiveInventory=1;
                skip;
                llGiveInventory=0;
        ::else ->
                llSetText=1;
                skip;
                llSetText=0;
        fi;
        event_request!255;
}
```

11. References

[http://secondlife.com/whatis/]

- *What is Second Life?*

[http://lindenlab.com/whitepapers/Escaping_Guilded_Cage_Ondrejka.pdf]

- *Cory Ondrejka 2003, Escaping the Guilded Cage, User Created Content and Building the Metaverse*

[http://www.anshechung.com/include/press/press_release251106.html]

- *Anshe Chung Becomes First Virtual World Millionaire*

[http://wiki.secondlife.com/wiki/LSL_Portal]

- *LSL Portal, LSL scripting references*

[http://spinroot.com/]

- *[http://spinroot.com/spin/Man/promela.html] Promela Man Pages*
- *[http://spinroot.com/spin/Man/ltl.html] LTL*
- *[http://spinroot.com/spin/Man/float.html] scope/limitations of SPIN*

[http://en.wikipedia.org/]

- *[http://en.wikipedia.org/wiki/Resident_(Second_Life)]Residents, Avatars*
- *[http://en.wikipedia.org/wiki/Virtualization] Virtualization*

[http://secondlife.com/knowledgebase/article.php?id=166]

- *How do I make an object "do" something? (Scripts)*

[http://openmetaverse.org/wiki/OpenSim]

- *Open Source Simulator project*

References on Model Checking and Related Work:
[Corbett, et. al 2000] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In Proceedings of the 22nd International Conference on Software Engineering, June 2000.

[Holtzman 2000] Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In Proc. of the 7th International SPIN Workshop, volume 1885 of LNCS. Springer-Verlag, September 2000.

[Havelund and Pressburger 1998]Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer, 2(4), April 1998.

[Demartini, Iosif, and Sisto 1999] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. Software - Practice and Experience, 29(7):577{603, 1999.

[Havelund and Visser 2002] K. Havelund and W. Visser, "Program model checking as a new trend", Int. J Softw Tools Technol Transfer 4(1) 2002 pp.:8-20

[Ball and Rajamani 2000] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Proc. Of the 7th International SPIN Workshop, volume 1885 of LNCS, pages 113{130. Springer-Verlag, September 2000.

[Xie and Browne 2002] Fei Xie and James C. Browne "Integrated State Space Reduction for Model Checking Executable Object-Oriented System Designs" Proceedings of FASE 2002 (Grenoble, France, April 2002) LNCS 2306 (Springer-Verlag, Berlin, 2002) pp. 64-80

[Xie, Levin and Browne 2002] Fei Xie,Vladimir Levin and Browne "ObjectCheck: A Modeling Tool for Executable Object-Oriented System Designs" Proceedings of FASE 2002 (Grenoble, France, April 2002) LNCS 2306 (Springer-Verlag, Berlin, 2002) pp. 331-335.

[Clarke, Grumberg and Peled, 2000] Edmund M. Clarke, Orna Grumberg and Doron A. Peled Model Checking: (MIT Press, Second Edition, 2000)

[Holtzman 2003] Gerard Holtzman The Spin Model Checker - Primer and Reference by, (Addison-Wesley 2003)

[Berard, et.al 2001]Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P. Systems and Software Verification Model-Checking Techniques and Tools (Springer-Verlag, 2001)

[Kurshan 1994] Robert P. Kurshan Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach (Princeton Series in Computer Science) (Princeton University Press, 1994)