

# **A Turn-based Strategy Game Testbed for Artificial Intelligence**

by

Joseph H. Souto

Presented to the Graduate and Research Committee of Lehigh University in  
Candidacy for the Degree of Master of Science

in

Computer Engineering

Lehigh University

April 2007

This thesis is accepted and approved in partial fulfillment of the requirements for the  
Master of Science.

---

**Date**

---

**Thesis Advisor**

---

**Co-Advisor**

---

**Chairperson of Department**

# Table of Contents

|   |    |
|---|----|
| List of Tables.....                             | iv |
| List of Figures.....                            | v  |
| Abstract.....                                   | 1  |
| Introduction.....                               | 3  |
| TIELT.....                                      | 4  |
| What is TIELT?.....                             | 4  |
| TIELT's Components.....                         | 6  |
| Sample TIELT Domain.....                        | 12 |
| Call to Power 2.....                            | 23 |
| What is Call to Power 2?.....                   | 23 |
| How to Play Call to Power 2.....                | 23 |
| Call to Power 2 API.....                        | 30 |
| Code Reference.....                             | 35 |
| The TIELT – Call to Power 2 Model.....          | 37 |
| What is the TIELT – Call to Power 2 Model?..... | 37 |
| Environment Model.....                          | 37 |
| Simulator Interface.....                        | 52 |
| Communication Protocol.....                     | 67 |
| Sample Session.....                             | 70 |
| Conclusions.....                                | 81 |
| Summary.....                                    | 81 |
| Future Work.....                                | 81 |
| Bibliography.....                               | 82 |
| Vita.....                                       | 83 |

# List of Tables

|   |    |
|---|----|
| <b>Table 1:</b> Call to Power 2 Action Models.....      | 44 |
| <b>Table 2:</b> Call to Power 2 Observation Models..... | 52 |
| <b>Table 3:</b> Call to Power 2 Outgoing Messages.....  | 59 |
| <b>Table 4:</b> Call to Power 2 Incoming Messages.....  | 67 |

# List of Figures

|   |    |
|---|----|
| <b>Figure 1:</b> Worst-case number of integrations.....                   | 6  |
| <b>Figure 2:</b> Best-case number of integrations.....                    | 6  |
| <b>Figure 3:</b> TIELT System Architecture.....                           | 7  |
| <b>Figure 4:</b> TIELT Decision System Architecture.....                  | 9  |
| <b>Figure 5:</b> Actions in TIELT.....                                    | 11 |
| <b>Figure 6:</b> Sensing Game State in TIELT.....                         | 12 |
| <b>Figure 7:</b> TIELT Environment Properties.....                        | 13 |
| <b>Figure 8:</b> TIELT State Variables.....                               | 14 |
| <b>Figure 9:</b> TIELT Classes.....                                       | 15 |
| <b>Figure 10:</b> TIELT Class Member Definition .....                     | 15 |
| <b>Figure 11:</b> TIELT Action Model.....                                 | 16 |
| <b>Figure 12:</b> TIELT Helper Function.....                              | 17 |
| <b>Figure 13:</b> TIELT Communication Properties.....                     | 18 |
| <b>Figure 14:</b> TIELT Outgoing Simulator Message.....                   | 19 |
| <b>Figure 15:</b> TIELT Incoming Simulator Message.....                   | 21 |
| <b>Figure 16:</b> TIELT Observation Model.....                            | 22 |
| <b>Figure 17:</b> Call to Power 2 Interface.....                          | 24 |
| <b>Figure 18:</b> Call to Power 2 Build Manager.....                      | 25 |
| <b>Figure 19:</b> Call to Power 2 City Improvements.....                  | 26 |
| <b>Figure 20:</b> Call to Power 2 City Manager.....                       | 27 |
| <b>Figure 21:</b> Call to Power 2 – Contact with Rival Civilizations..... | 28 |
| <b>Figure 22:</b> Call to Power 2 Combat.....                             | 29 |
| <b>Figure 23:</b> TIELT – CTP2 Communication Protocol.....                | 69 |
| <b>Figure 24:</b> Start of Enhanced Session.....                          | 70 |
| <b>Figure 25:</b> First Turn with Enhanced Session.....                   | 71 |
| <b>Figure 26:</b> Building a City.....                                    | 72 |
| <b>Figure 27:</b> Producing Military Units.....                           | 73 |
| <b>Figure 28:</b> New Units Completed.....                                | 74 |
| <b>Figure 29:</b> Exploring the Map.....                                  | 75 |
| <b>Figure 30:</b> Building City Improvements.....                         | 76 |
| <b>Figure 31:</b> Defending a City.....                                   | 77 |
| <b>Figure 32:</b> Attacking an Enemy City.....                            | 78 |
| <b>Figure 33:</b> Attack Success.....                                     | 79 |
| <b>Figure 34:</b> Attacking Enemy Unit.....                               | 80 |

## **Abstract**

The DARPA Transfer Learning Project is a program that intends to develop, implement, demonstrate and evaluate theories, architectures, algorithms, methods, and techniques that enable computers to apply knowledge learned for a particular, original set of tasks to achieve superior performance on new, previously unseen tasks. [Transfer Learning, 2007] Specifically, the goal is to develop and evaluate the performance of transfer learning agents in simulated environments.

The Testbed for Integrating and Evaluating Learning Techniques (TIELT) is a software tool developed by the Naval Research Lab that is used to integrate AI systems with gaming simulators, and evaluate how well those systems learn on selected tasks. [Aha, 2007]

Call to Power 2 is a turn-based strategy game based on Civilization. In Call to Power 2, the player develops a civilization and progresses through technological ages from ancient to near-future times. During the game, the player interacts with other rival civilizations and manages the resources and economy of their civilization. It is a good game to use to test intelligent agents because there are many disparate factors that much be gauged at any given time, and decisions must be made based upon these elements. Some of the decisions a player or intelligent agent must make include: where to build new cities, whether to explore or expand the civilization, how to dispose forces to attack enemy units, and which technological advances to seek.

As part of the Transfer Learning Project, intelligent agents will be tested in conjunction with Call to Power 2. To this end, Call to Power 2 needed to be integrated with TIELT to use and evaluate these agents. This integration required effort on two

fronts. First, the game world and important game objects had to be modeled in TIELT. Second, an API needed to be developed that both enables simple control of the actions of the computer player in Call to Power 2, as well as allowing an interface with TIELT so that a TIELT agent can execute the functions available in the API. This thesis explains the work entailed in this process and shows an example of the end results.

# 1. Introduction

Transfer learning is a theory proposed by Thorndike & Woodworth (1901) that proposes that things humans approach new tasks by applying things learned in other similar situations. [Thorndike & Woodworth, 1901] A simple transfer learning example would be that when a human drives a bus for the first time, they will apply things that they have learned about driving when they drove a car.

The DARPA Transfer Learning Project is based on the belief that artificial intelligent agents can be constructed and improved by the same method. By training an intelligent agent one context, in our case a computer game, the agent should then perform better in other similar types of computer games.

One of Lehigh's tasks in the Transfer Learning Project is to enable the testing of intelligent agents in the turn-based strategy game Call to Power 2. Call to Power 2 was chosen for two reasons. First, it is a game that involves many strategic factors that an agent would have to make decisions on. Some of the many decisions a player or intelligent agent must make include: where to build new cities, whether to explore or expand the civilization, how to dispose forces to attack enemy units, and which technological advances to seek. Second, the source code for Call to Power 2 was made open-source by Activision in 2003, and is maintained by an open-source community called Apolyton [Apolyton, 2007]. This allows us to make any modifications to the game necessary to enable integration with the AI tools used by the Transfer Learning Project.

The results of this project will not only be of benefit to the DARPA Transfer Learning Project. The development of the API will be of great benefit to other AI researchers because it provides a simple interface that will allow them to construct and



test intelligent agents for use in Call to Power 2 without requiring in-depth knowledge of the details of the Call to Power 2 system architecture.

There were a few difficulties involved with doing this project. First, even though Call to Power 2 has been made open-source, for legal reasons Activision had to remove all documentation from the code. This makes it rather challenging to come to an understanding of the Call to Power 2 system architecture and the design decisions that the developers made along the way. Consequently, it can be very arduous to add additional features to the API as it requires understanding some of the internal functionality of the game in order to interact with it. Additionally, TIELT is a continually evolving system to meet the needs of the Transfer Learning Project. When new versions are released by the Naval Research Lab, the work done to integrate a game with previous versions of TIELT must be converted to the new versions.

This thesis is divided as follows. Section 2 introduces the AI research tool, TIELT. Section 3 describes Call to Power 2 and our API. Section 4 explains the TIELT model of Call to Power 2. Section 5 shows a sample session of Call to Power 2 played with the completed TIELT integration. Lastly, Section 6 lists conclusions and further work possible due to these efforts.

## 2. TIELT

### 2.1 What is TIELT?

Tielt is a software tool developed by the Naval Research Lab (NRL) to facilitate AI researchers in the evaluation of intelligent decision systems in various game engines (e.g., real-time strategy, discrete strategy, role playing, team sports, first-person shooter). Emphasis has been placed upon games that are applicable to military simulations, which is part of why Call to Power 2 was selected in conjunction with the Transfer Learning Project. [Aha, 2007]

One might ask why is TIELT necessary at all? Given that Call to Power 2 is open-source, why not simply modify the existing AI engine for the computer player directly? The answer to these questions is time and integration complexity. Even though a game may be open-source, it would still be a daunting task to come to a complete enough understanding of the system architecture to be able to modify the AI code directly. Additionally, modifying the AI directly would make it difficult to test the performance of more than one algorithm for an intelligent agent. Since your newly developed code would be very tightly bound to the architecture of the game, it would require almost a complete rewrite of the code to test a different type of agent program.

As for complexity, given that the purpose of the transfer learning project is to test a variety of intelligent agents across a variety of simulators, for each one of  $x$  agents you would need to perform an integration for each one of the  $y$  simulators you want to test the agent on. This requires a total of  $x*y$  integrations. TIELT, however has the advantage of acting as an intermediary between the simulators and the agent programs. Once a simulator is integrated with TIELT, it does not need to know how to communicate with any of the agent programs because it TIELT hides the details of the communication from

the simulator. Likewise all the agents need to know is how to communicate with TIELT; they do not need to know how the communication is performed with the simulation engine. Therefore, with the aid of TIELT, the number of integrations required to test a suite of intelligent agents is reduced from  $x*y$  to  $x + y$ . See the figures below.

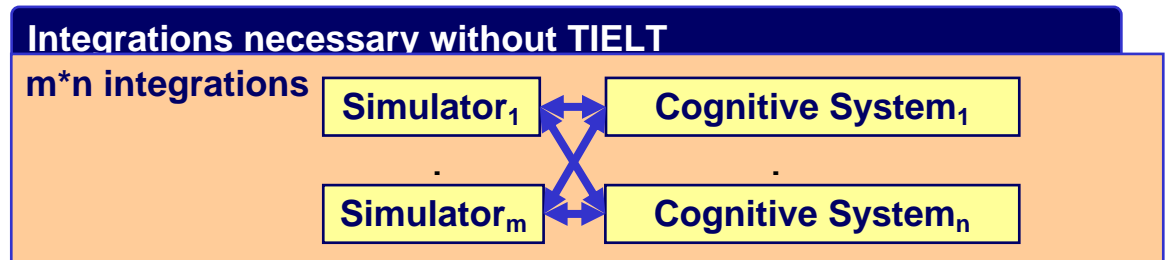


Figure 1: Worst-case number of integrations. Taken from: [Aha, 2004]

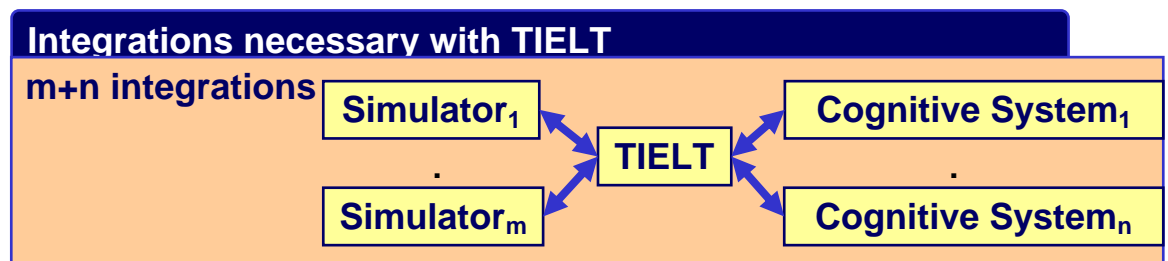


Figure 2: Best-case number of integrations. Taken from: [Aha, 2004]

## 2.2 TIELT's Components

By acting as this intermediary between the simulation engine and the cognitive system, the task of evaluating a transfer learning agent becomes broken down into a series of much simpler components. The following diagram shows the resulting system in Figure 3.

## TIELT: Integration Architecture

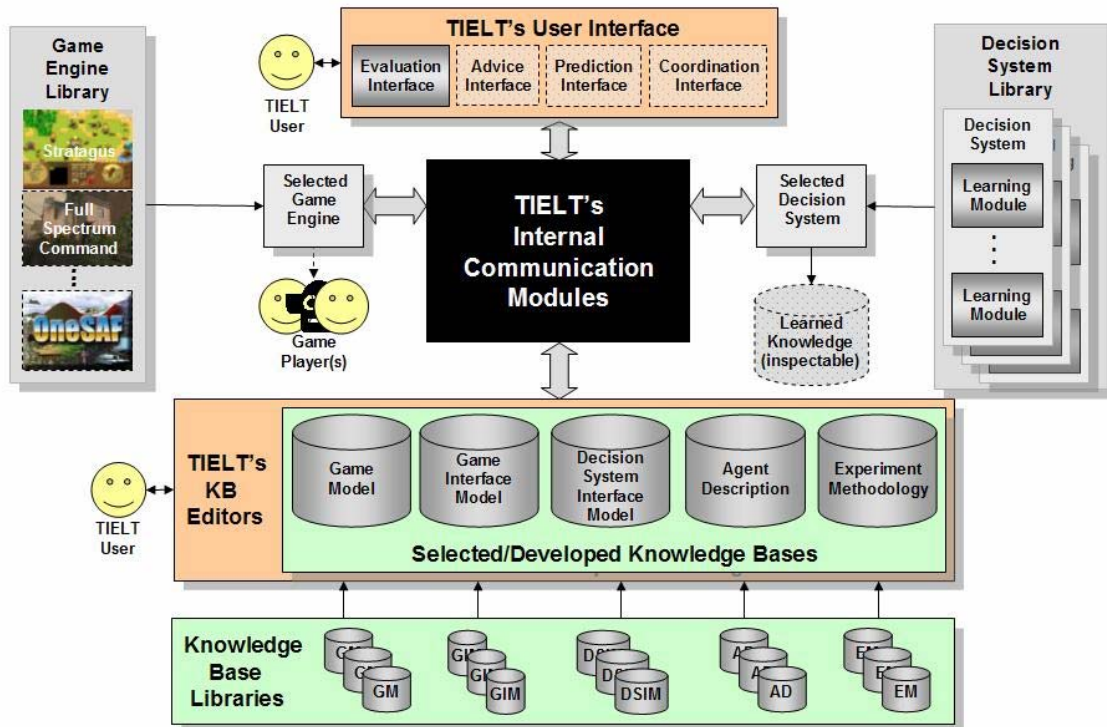


Figure 3: TIELT System Architecture. Taken from: [Aha, 2007]

Before the experiments can begin, one must first build the game model for the simulation engine in TIELT. This can be seen at the bottom of figure 3 above in which the user of TIELT creates the game model, game interface model, decision system interface model, etc. These components will be explained in greater detail in Section 2.3.

Once TIELT has been integrated, experiments can be performed. As seen in Figure 3, on the leftmost end of the system, we have the game that we're testing our agent with. The game sends information about the changing state of the game world to the TIELT module. Through TIELT's user interface, the user of TIELT can observe this information about changes to the state of the world. When these messages are received, internal state information within TIELT is then updated according to rules set forth by the

TIELT user. These messages about these changes to the game state are also passed on to the decision system that has been integrated with TIELT, as seen in Figure 3 on the right side. The decision system uses these sensors combined with the current state information within TIELT's knowledge bases to evaluate what is happening within the game world. Once it makes a decision on which action to perform, it sends message(s) with the proposed action(s) back to TIELT. These actions can be used in one of two ways. The TIELT user can observe these proposed actions, and either use the agent's proposed actions as advice system to play the game in a more informed manner. Or the agent can control the actions of the player directly by sending these action messages through TIELT to the simulation engine. The TIELT user can also observe the actions that the decision system has performed in order to monitor the quality of the decisions the agent program has made.

It can be seen therefore, that TIELT makes it very easy to test and compare the performance of a large variety of intelligent agents in a given simulation engine. To test the performance of agent program Y instead of agent program X, all that is required is to have a TIELT integration with each agent program. Then, given a suite of tests in the game environment, all which is required is to run these tests with each different agent program hooked up at the other end of TIELT.

To get a more in-depth view of how games are played in conjunction with TIELT, consider the following example situation for Call to Power 2. Suppose you have a Settler, and want to settle a new city. Assume that you have queried the game engine about whether a city can be build on the tile that unit is currently on (more specifics on how this task is done will be provided later in Section 4). TIELT receives a response

message, *CityBuildable*, indicating that a city can be built on that tile. The Learning Translator TIELT module sees that the game engine has indicated that a city can be built on that tile and decides to tell the game engine to have that Settler build a city. In order to perform this action, it creates a message, *BuildNewCity*, which contains parameters indicating that this Settler should build a city on its tile. After it has been created, the Learning Translator module sends this *BuildNewCity* message to the Decision System. Once the Decision System has received the *BuildNewCity* message, it sends it to the Action Translator TIELT module to convert this desire to settle a new city into an output message to be sent to the game engine. This whole process is summarized in Figure 4 below.

### Fetching Decisions from the Decision System (City placement example)

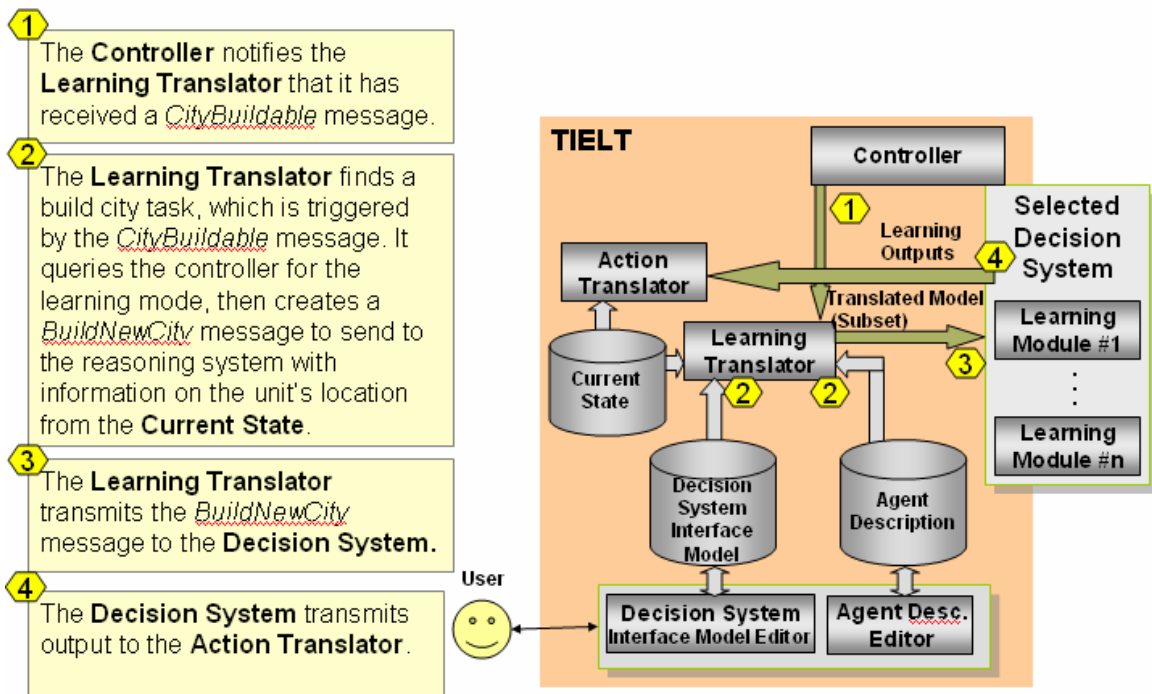
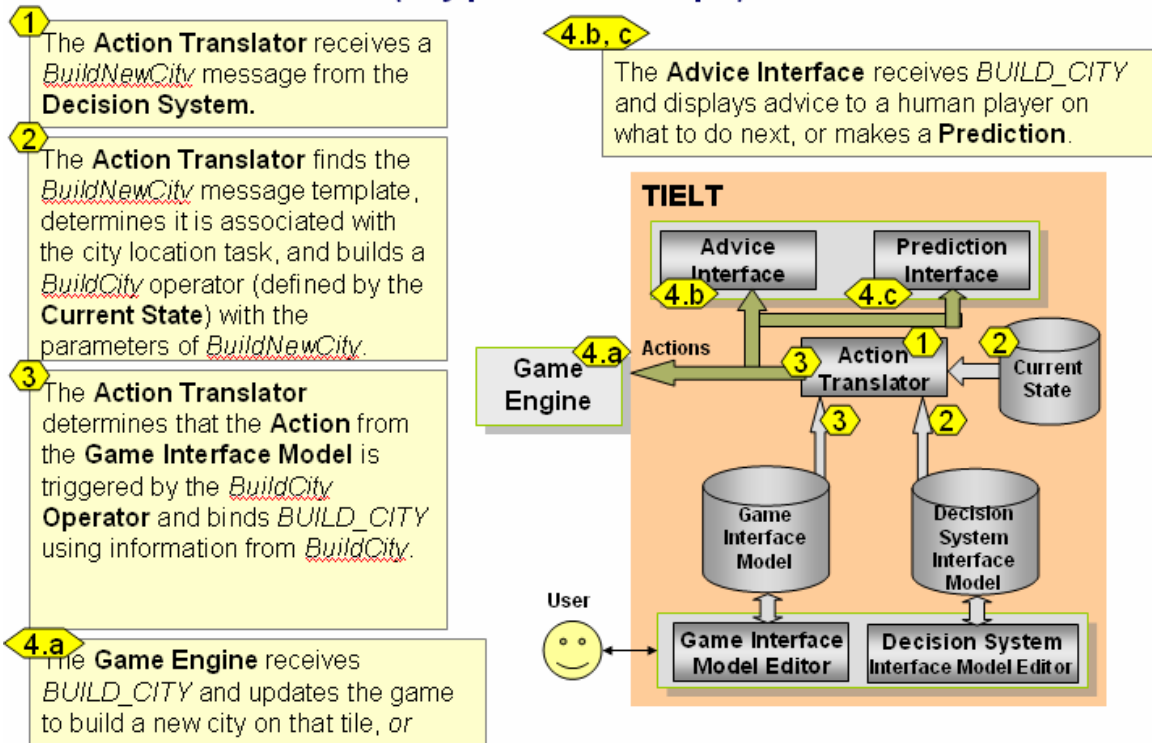


Figure 4: TIELT Decision System Architecture. Taken from: [Aha, 2007]

After the decision has been made, and the desired action has been selected, the next step is for the Action Translator TIELT module to convert the action to a message that will tell the simulator what to do. When the Action Translator receives the *BuildNewCity* decision from the Decision System, it determines that it is associated with a particular operator for interfacing with the simulation engine. In this case, it associates the *BuildNewCity* decision with an action model called *BuildCity* and populates the parameters of the *BuildCity* operator with information from *BuildNewCity* and the current state of the game as kept in TIELT. The *BuildCity* action model is shown to the TIELT user, and is then using information from the Game Interface Model module, TIELT associates the *BuildCity* action model with a message, *BUILD\_CITY*, which when populated with the appropriate parameters is sent out on a socket to the game in order to perform the action of building a city. Finally the game receives the *BUILD\_CITY* message and has the Settler create a new city. This set of actions is shown in Figure 5 below.

# Acting in the Game World

(City placement example)



**Figure 5: Actions in TIELT. Taken from: [Aha, 2007]**

After the *BUILD\_CITY* message has been received by the game engine, the Settler will build a new city. Since the creation of a new city changes the game state, once this action has been completed, the game engine will send a message, *CITY\_SETTLED*, back to TIELT indicating where the new city was built. The Game Interface Model will see the *CITY\_SETTLED* message and translate it to its corresponding Observation Model, *CitySettled*. The parameters of this observation model are then used, along with the rules in the Game Model, to determine the manner in which TIELT's model of the game state should be modified when *CitySettled* has been received. After the current state has been updated by the Model Updater TIELT module, it informs



the Controller that a *CitySettled* event has been perceived, and passes this information on to the user and the agent program. This process can be seen below in Figure 6.

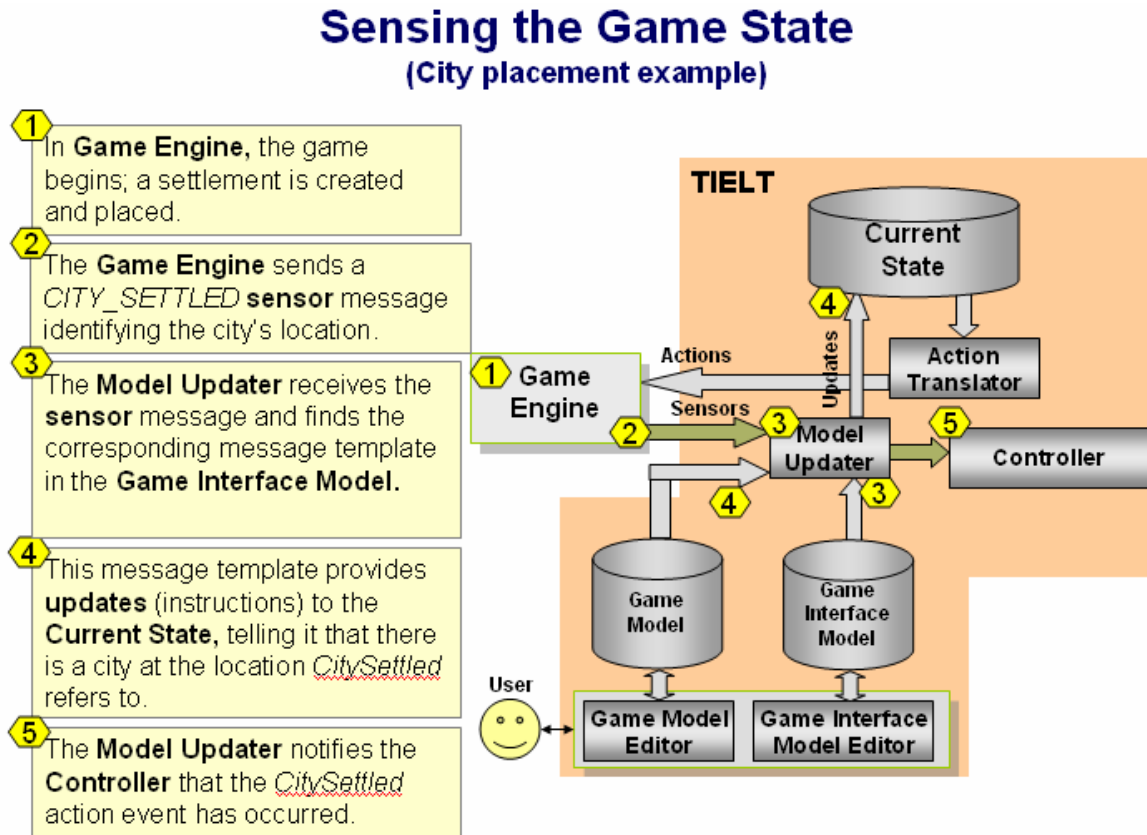


Figure 6: Sensing Game State in TIELT. Taken from: [Aha, 2007]

### 2.3 Sample TIELT Domain

With all that in mind about the architecture of TIELT, now we can discuss how to model the rules of a game within TIELT. We will do this by example. We will show how a simple domain is set up by giving a brief overview of how to model a well known game, Chess, in TIELT.

The first step with TIELT is to set up the Environment Model. Here we define a few rules about what type of game we wish to model and how it works.

**Properties of Environment**

Field:  Time:

Accessibility:  Inaccessible  Accessible

Determinism:  Deterministic  Stochastic

Transition Knowledge:  Incomplete Rules  Complete Rules

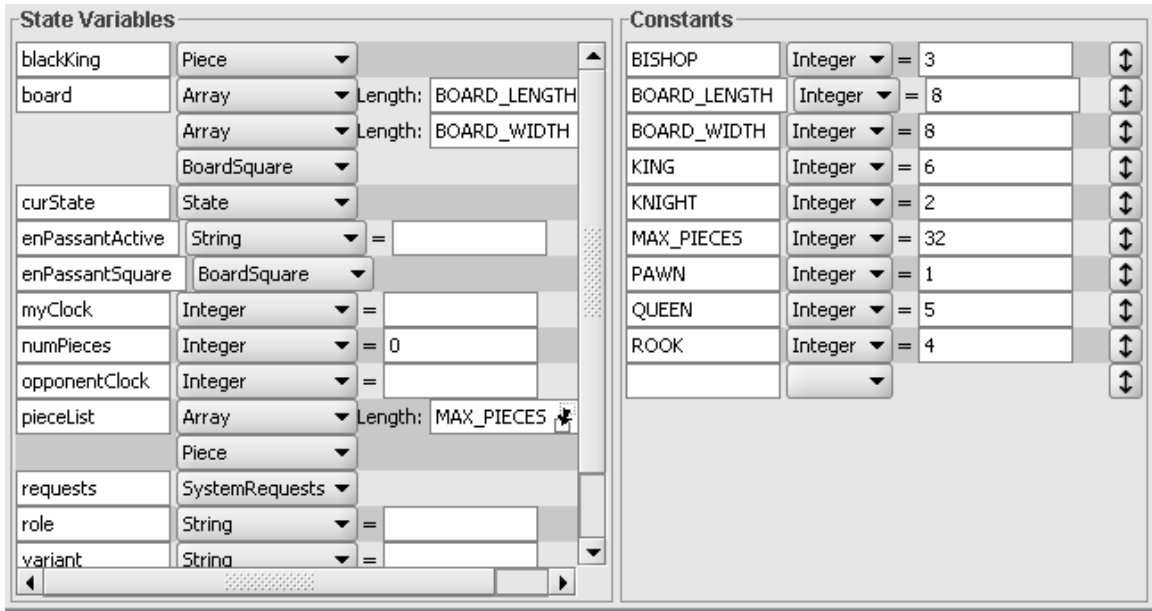
Competitiveness:  Single Player  Zero Sum  Non-Zero Sum

Cooperativeness:  Team-Oriented  Non-Team-Oriented

**Figure 7: TIELT Environment Properties**

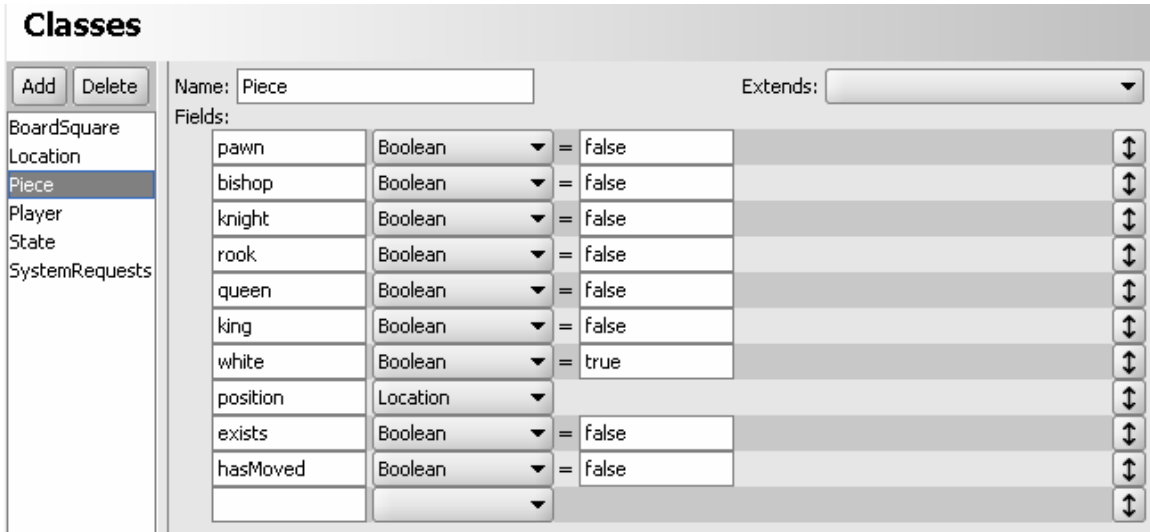
Chess is a game played on a 2D board in which two opposing players take turns making their moves. At any given time, the entire game world is completely visible to either of the players (accessible). The next state of the game is determined entirely by the actions of one of the players (deterministic). All the rules of the game are known (complete). And only one of the two players can win the game (Zero Sum, Non-Team-Oriented).

After defining the rules of the game, the next step is to define the state variables. These are variables which represent data about the current state of the game world. When TIELT perceives a change in the state of the game world, the state variables will be modified in order to reflect this change in the game world. Examples of state variables one would want to use in Chess include a data structure to represent the board, the number of pieces on the board, and a representation of each different type of piece used in the game. These, among other sample state variables are shown in Figure 8 below.



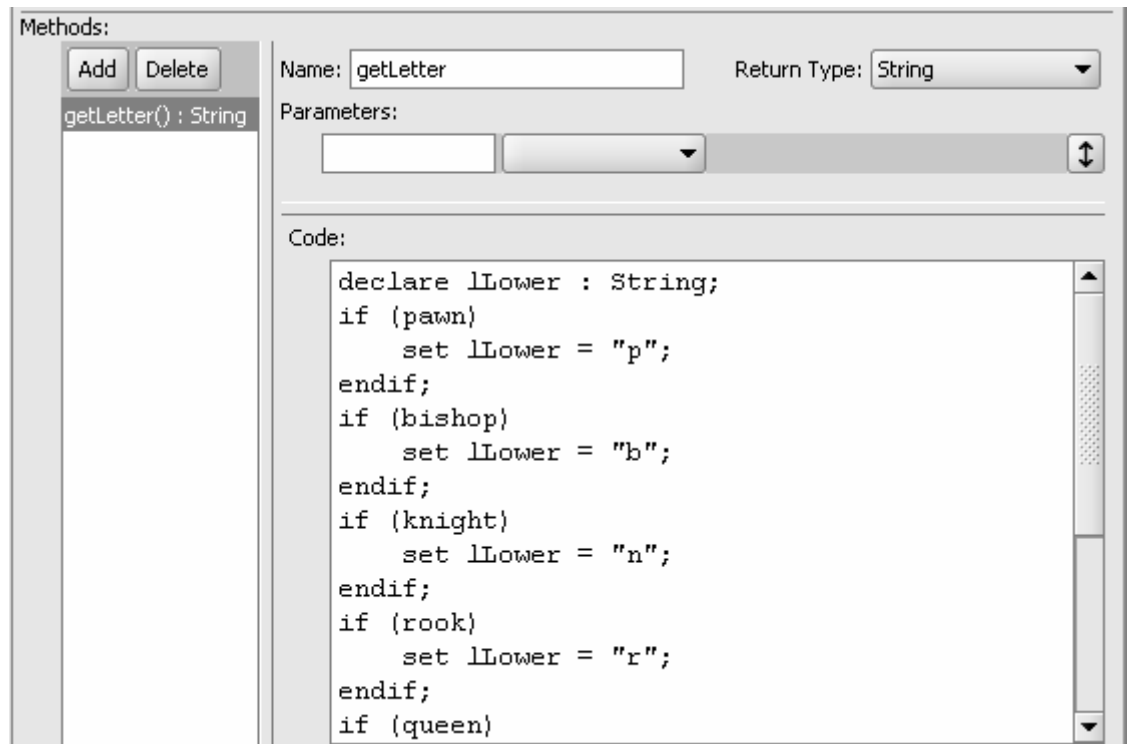
**Figure 8: TIELT State Variables**

Corollary to specifying the state variables is defining various classes to be used in TIELT. Classes in TIELT are much like C++ classes in that they have members, that can be either primitive data types or objects, and functions which operate upon those member variables. An example of a class used for Chess is the Piece class. The Piece class contains such information as which type of piece something is, what player it belongs to, and where it resides on the board among other information. The member variables of the Piece class are shown in Figure 9.



**Figure 9: TIELT Classes**

The Piece class also has one member function, called `getLetter()`, which is used to return a single letter representing the type of piece a Piece object represents. This member function is used by other functions to get a quick reference to the types of pieces being dealt with. Part of the definition of the `getLetter()` function is shown in Figure 10.



**Figure 10: TIELT Class Member Definition**

After these aspects of the state are defined, the next step is to specify the Action Models. As shown earlier, an Action Model is a TIELT representation of an action or set of actions possible in the game world. One such action model needed for Chess is something to represent moving a piece on the game board. This action model can be defined simply as movePiece(), which takes arguments defining what piece is to be moved, and what location on the board it is to be moved to. Figure 11 shows how the movePiece() message is declared in TIELT.

**Action Model Entry**

Name:

Parameters

|                                  |                                       |   |
|----------------------------------|---------------------------------------|---|
| <input type="text" value="p"/>   | <input type="text" value="Piece"/>    | ↕ |
| <input type="text" value="lTo"/> | <input type="text" value="Location"/> | ↕ |
| <input type="text"/>             | <input type="text"/>                  | ↕ |

Relevant Phases:

- OpponentTurn
- MyTurn

Capable Roles:

Add Phase ...

Behavior Rule:

Conditions:

State Changes:

= ;

Validate    Type Code ?

Transition Table:

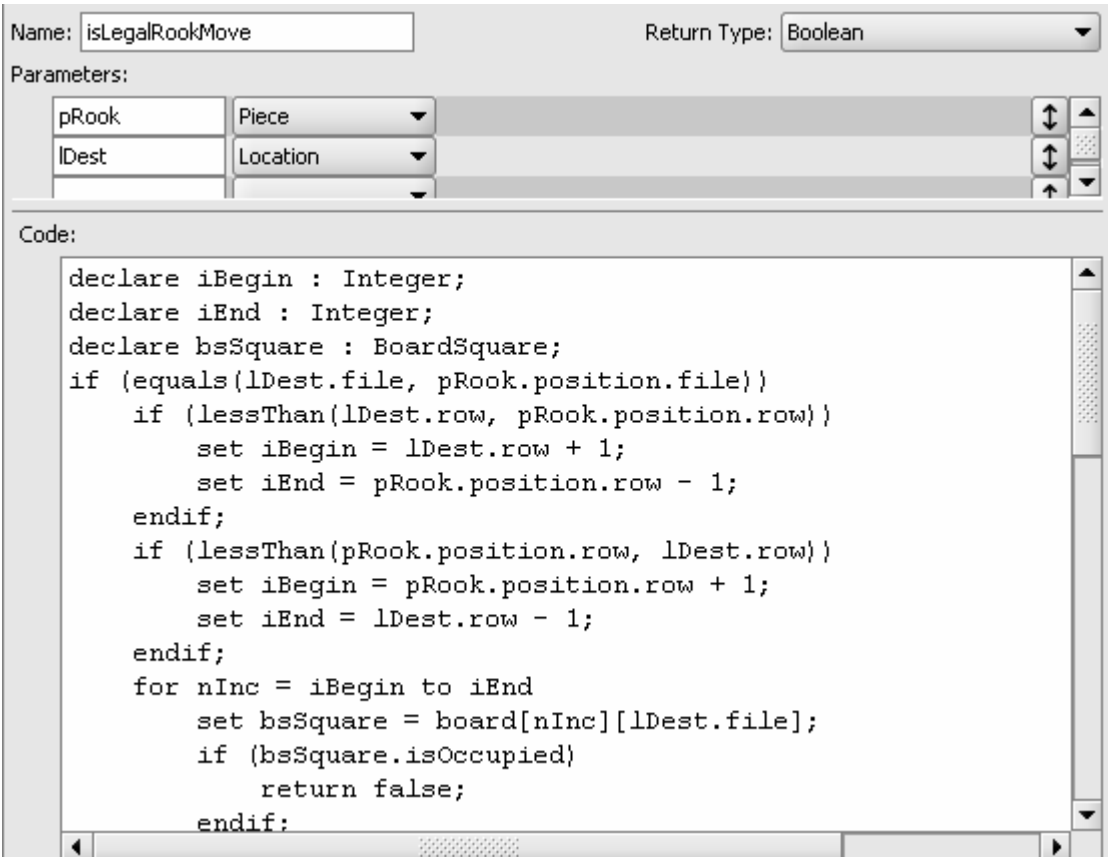
|                             |   |  |   |
|-----------------------------|---|--|---|
| Order: 0                    | Next Phase: <input type="text" value="MyTurn"/> | Condition: <input type="text" value="true"/> | ↕ |
| Order: <input type="text"/> | Next Phase: <input type="text"/>                | Condition: <input type="text" value="true"/> | ↕ |

Save    Cancel

**Figure 11: TIELT Action Model**

As seen, the Action Model is defined by its name, the parameters it is called with, what state of the game it can be used in, whether there are any specific preconditions restricting the use of that action (“Conditions”), whether the action will change TIELT’s game state (“State Changes”), and if this action will move the game into a new phase.

It will also be helpful to define some helper functions to perform logical operations. These functions will be useful for things like determining if a move can be made, and what location on the board a moved piece will be at as a result of a move. An example of a helper function is shown below in Figure 12.



```
isLegalRookMove
Return Type: Boolean
Parameters:
pRook Piece
lDest Location
Code:
declare iBegin : Integer;
declare iEnd : Integer;
declare bsSquare : BoardSquare;
if (equals(lDest.file, pRook.position.file))
  if (lessThan(lDest.row, pRook.position.row))
    set iBegin = lDest.row + 1;
    set iEnd = pRook.position.row - 1;
  endif;
  if (lessThan(pRook.position.row, lDest.row))
    set iBegin = pRook.position.row + 1;
    set iEnd = lDest.row - 1;
  endif;
  for nInc = iBegin to iEnd
    set bsSquare = board[nInc][lDest.file];
    if (bsSquare.isOccupied)
      return false;
    endif;
  endfor;
endif;
```

**Figure 12: TIELT Helper Function**

Here we see an example helper function for determining if a rook’s move is valid. Given as input a piece object referring to a rook, and a location object referring to the rook’s intended destination, isLegalRookMove() will return true if the move is valid for

the rook and false if the move is invalid. Functions like these are used in various places to assist in deciding what to do, as we shall see shortly.

After defining all the necessary Action Models, we will now take a step away from the Environment Model for the moment, and begin setting up the Simulator Interface Model. The first thing to do is define how TIELT will communicate with the game engine. The simplest way to do this is via a TCP/IP connection on a socket. A format specifying precisely how messages and their associated parameters will be sent will also be defined here.

Chess.xml: Action Models x Chess.xml: Observation Models x CTP\_ver1.xml: Communication x

Method: TCP/IP

Hostname: 127.0.0.1 Port #: 6666  Act as a server

In the formatting boxes below, the following substitutions apply:  
%mn - message name  
%pn - parameter name  
%pv - parameter value  
Spaces are important. If you include a space in the spec, TIELT will expect it in the message, and the message will not be read properly without it. However, spaces will be tolerated between values in messages even if not present in the spec. In order to put spaces in string values, please specify string delimiters.

This header will be used once per message.  
%mn(  
This will be repeated for every parameter in the message.  
%pv  
This will separate parameters.  
,  
This closer will be used once per message.  
)  
This message separator will be used to determine whether a message is complete.  
\r\n  
This format will be used for empty messages (with no parameters).  
A second format cannot be validated without a message separator.  
%mn  
These string delimiters can be used to enclose strings with arbitrary punctuation.  
Start: " End: "  Quotes may be nested:  
Example: foo(1.0,false,"Hello World!")  
If necessary, provide formats for transmission of arrays and objects.

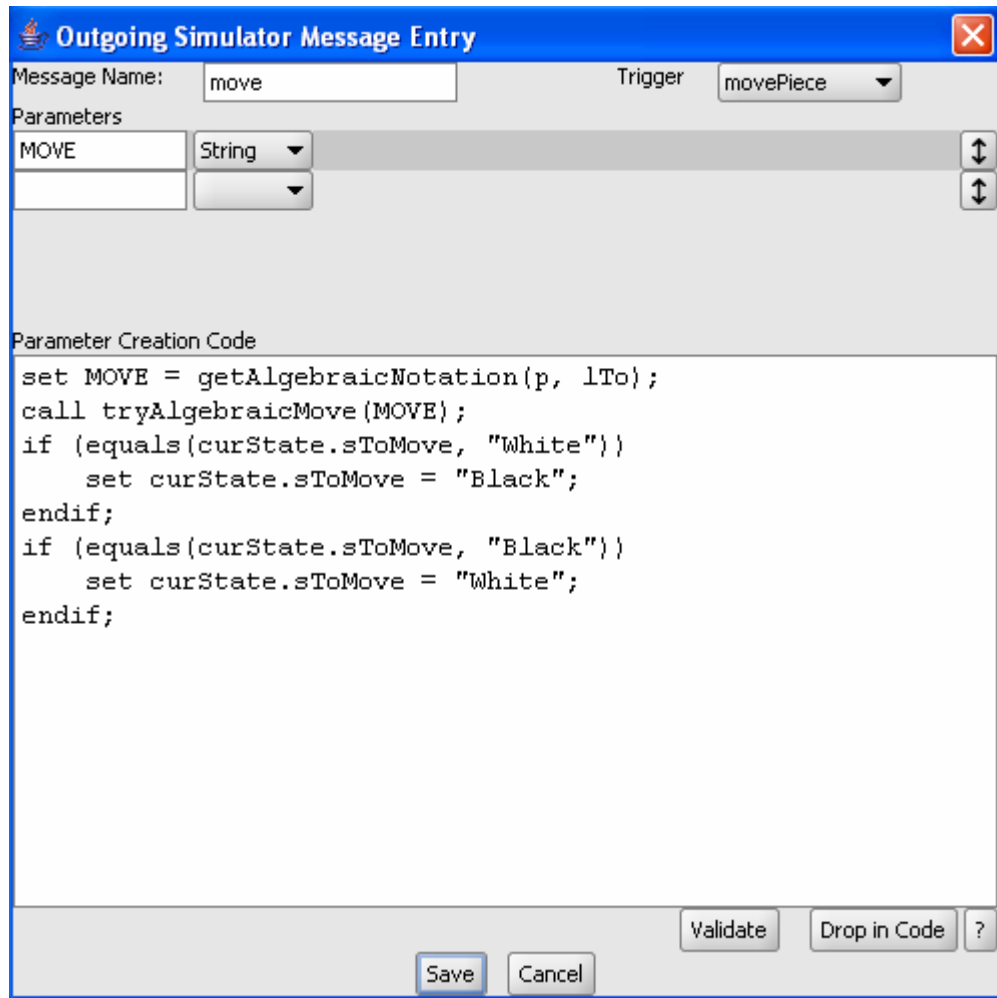
Array Format ... Object Format ...

**Figure 13: TIELT Communication Properties**

As seen above, messages to the game engine have been defined to begin with the name of the message, followed by the set of comma-delimited parameter values in parenthesis. Messages can be defined in many ways, but this format is simple, and intuitively looks similar to function calls in a programming language.

The next step is to define the messages that will be sent to the game engine in order to cause things to happen within the game. To continue with our example, we will define a message to move a piece on the board, which we will call `move()`. This message will be triggered whenever TIELT invokes the `movePiece()` action model, and thus it will have access to `movePiece`'s parameters, `p` and `lTo`. The definition of `move` is shown in Figure 14.



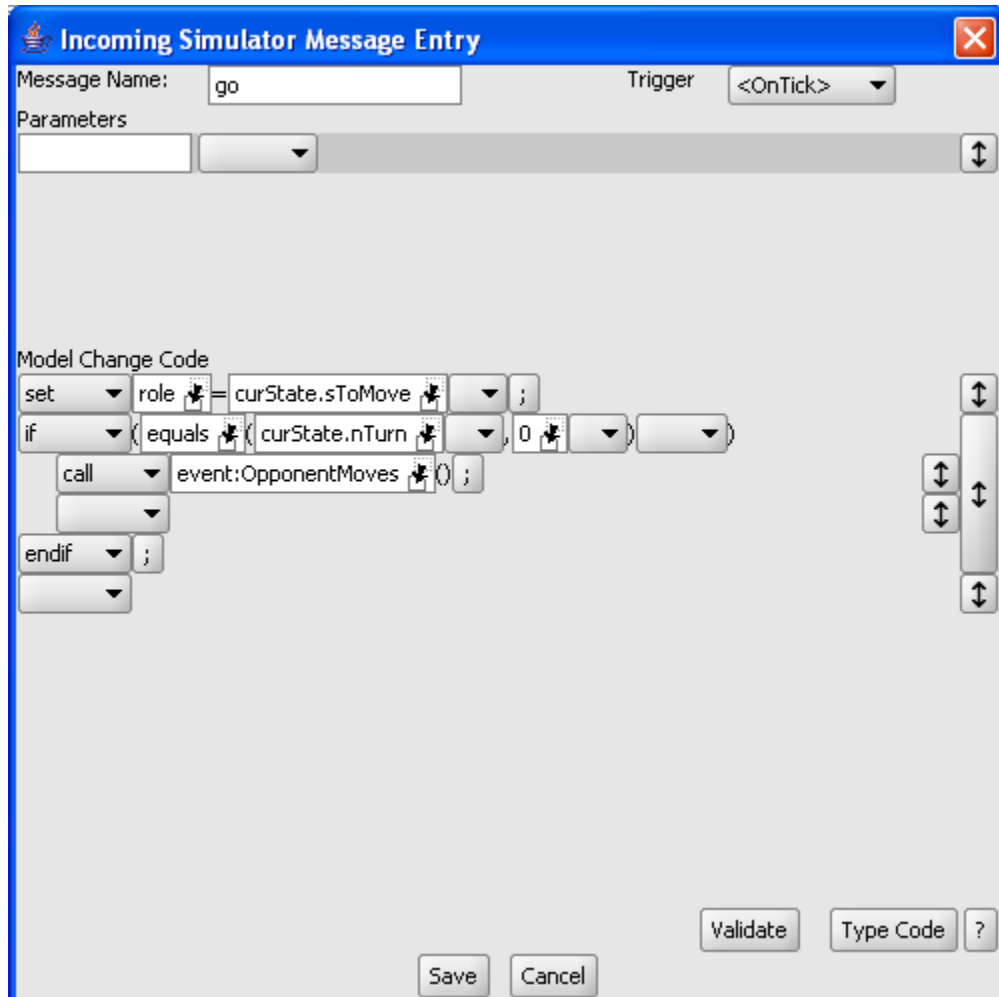


**Figure 14: TIELT Outgoing Simulator Message**

Here we see that move will send a string to the game engine to represent what piece it wants to move to which location. In order to set up this string, it uses some of the helper functions mentioned earlier in order to convert the TIELT objects for p and lTo into a string type. Then it checks which player made this move in order to update the game state to reflect the player whose turn will be next after this move is made.

After all these outgoing messages have been defined, we must define messages coming in from the game engine to TIELT. These messages are sent from the simulator whenever there is a change in the game world. In the Chess example, one such incoming

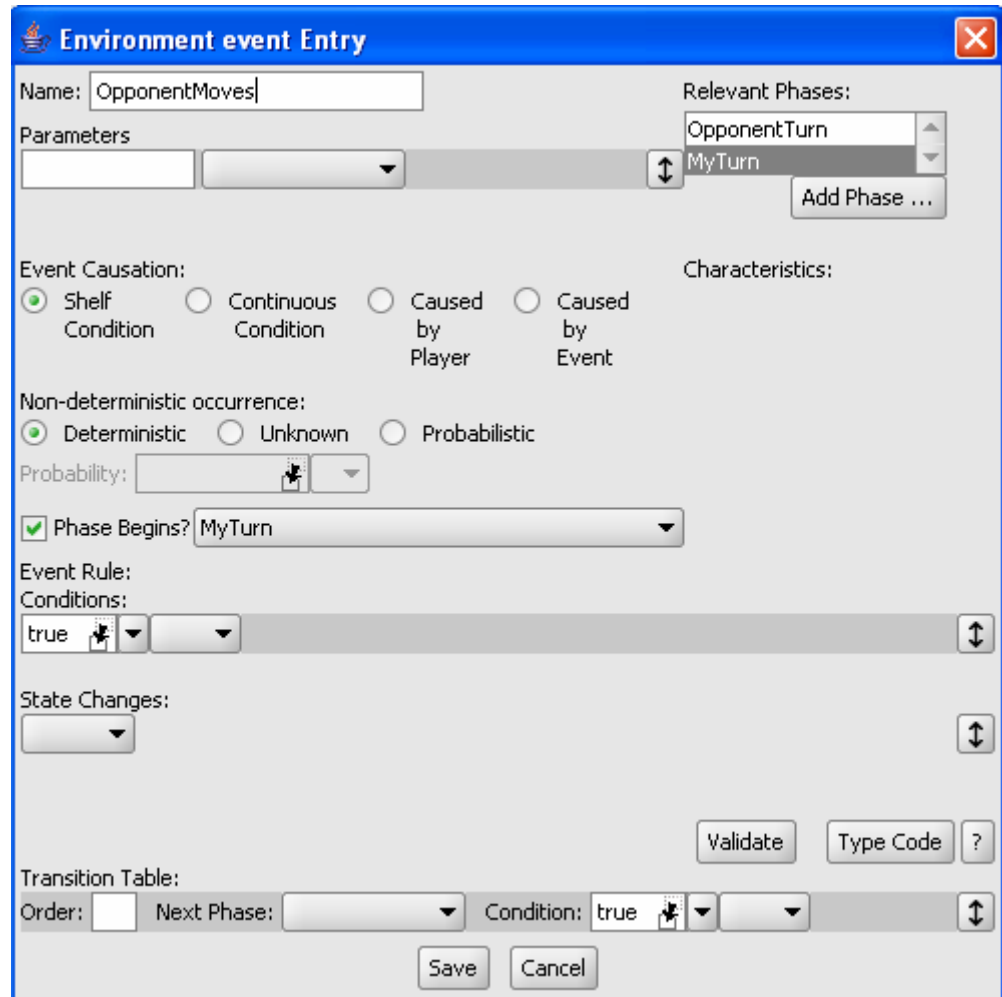
message would be a message representing the end of the opposing player's turn, which we will call go(). Upon receipt of the go() message, the game state will be updated to reflect that it is now our turn. The definition of the go() message is shown below in Figure 15.



**Figure 15: TIELT Incoming Simulator Message**

After all of the incoming messages have been defined, the final step is to complete the environment model by setting up the Observation Models. Observation Models represent our ability to sense changes in the game world. One such observation

we will want to sense is associated with the go() message from Figure 15. As seen in the model change code for go(), it triggers the OpponentMoves observation model. When this happens, we will want to update the game state in order to reflect the fact that it is now our turn. This can be seen in Figure 16 below, in that when OpponentMoves is triggered it begins the phase “MyTurn”.



**Figure 16: TIELT Observation Model**

Once all the sensors are defined by the Observation Models, we have completed our TIELT interface with the desired game engine.

## **3. Call To Power 2**

### **3.1 What is Call to Power 2?**

Call to Power 2 is a turn-based strategy game based on Civilization. In Call to Power 2, the player develops a civilization and progresses through technological ages from ancient to near-future times. During the game, the player interacts with other rival civilizations and manages the resources and economy of their civilization. Victory is attained by either military conquest of all rival civilizations, or by reaching world peace through diplomatic means. It is a good game to use to test intelligent agents because there are many disparate factors that must be gauged at any given time, and decisions must be made based upon these elements. Where to build new cities, whether to explore or expand the civilization, how to dispose forces to attack enemy units, and which technological advances to seek are but a few of the many decisions to be made by a player or an intelligent agent

### **3.2 How to Play Call to Power 2**

A game of Call to Power 2 begins with the player in control of two Settler units. Settlers are non-military units that build new cities. At the start of the game, the player chooses where to build their initial cities. Things to consider when choosing where to place a new city are the type of terrain it is on, the amount of food and commerce that tile can provide, and whether there are any tradable goods in the vicinity of that tile. See Figure 17 below for a view of this situation at the start of the game.



**Figure 17: Call to Power 2 Interface**

Here we see the portion of the map we start the game in. As seen, the board is divided into square tiles, and is viewed from overhead. Tiles that have not yet been explored are displayed in black. There is one Settler built a city on a plains tile. Within the bounds of that city (dashed white line) is a tradable good, tobacco. The other Settler we started the game with is on a forest tile near another tradable good, beavers. In the bottom left is a minimap displaying the explored portion of the map in relation to the rest of the unexplored tiles. In the bottom right is a control panel allowing access to the information about various aspects of your civilization.

Once a city has been built, units can be produced within that city. Military units are one example of a type of unit that can be built in the city. In the figure above, we can

see that a Warrior is being built in Rome, and will be completed in 9 turns. The production of units in a city is controlled via the Build Manager window for each city.

An example of the Build Manager window can be seen below in Figure 18. .



**Figure 18: Call to Power 2 Build Manager**

The Build Manger shows all the units that can be built at that point (more become available as the civilization advances to new technological ages), and has a build queue which allows multiple units to be built sequentially after the previous unit in the queue has completed.

Units are not the only type of thing that can be built in a city. Another type of thing that can be produced is a city improvement. City improvements do not appear on the map, but are stored internally in the state of each particular city. City improvements provide various types of benefits when built, including increased commerce or production, and additional defense against attacking enemy units. An example of a city improvement can be seen below in Figure 19 where we see that building a Granary would

provide a bonus to the production of food within this city. Another type of thing that cities can produce is a Wonder. Wonders are accomplishments that take a long time to build and provide a bonus to your entire empire upon their completion. Examples of Wonders include the Great Wall of China, which removes all barbarians from your empire, and Galileo's Telescope, which increases the rate at which new science is learned.



**Figure 19: Call to Power 2 City Improvements**

Many other things can be managed within each particular city. See Figure 20 below for a sample of some of the other many aspects of a city. The details of what each specific thing in Figure 20 represents are beyond the scope of what was done for this project, but they are shown to provide an idea of the amount of micromanagement available in the game. It should be noted that the game manages the growth of a city automatically unless the player tweaks the parameters for its management. Additionally, a “Mayor” can be assigned to the city, as seen in Figure 20, which automatically

optimizes the resources of a city towards a particular goal (e.g.: Production, Science, etc.).



**Figure 20: Call to Power 2 City Manager**

As the game progresses, the player eventually comes into contact with rival civilizations. Figure 21 shows a screenshot from later in the game after a few military units have been produced and more of the map has been explored. As seen, two cities belonging to a rival civilization have been discovered, Swansea and Cardiff, indicated by the orange number next to them.





**Figure 21: Call to Power 2 – Contact with Rival Civilizations**

Call to Power 2 offers three methods of obtaining victory: diplomacy, scientific advancement, and military conquest. Diplomacy involves offering resources or scientific advances to the other civilization in exchange for eventually attempting to forge treaties and eventually a military alliance with that civilization. Victory through science entails attaining each scientific advancement in the tech tree, and then building the Solaris Project wonder. Diplomacy and scientific advancement are rather complex to deal with from an agent's perspective, and is thus beyond the scope of this project, so it will not be discussed further. Military conquest involves destroying all existing enemy units and cities with your military units. In the bottom right corner of the figure above you can see

the actions available with the highlighted Warrior unit. Military units have the ability to attack enemy units or enemy cities.

When an attack command is issued, combat with the defending enemy units is initiated. An example of a combat window is shown in Figure 22 below. There we see the highlighted Warrior from the previous picture attacking the city adjacent to it, which is defended by an enemy Hoplite unit. Combat is conducted by each unit striking the other simultaneously and doing an amount of damage within a variable range according to which type of unit it is. Additionally, there are bonuses given to the defender when they are garrisoning a city. Combat continues until either the attacking player decides to retreat, or when one of the player's units is wiped out.



**Figure 22: Call to Power 2 Combat**

### **3.3 Call to Power 2 API**

The previous section showed a few of the many complexities involved in playing a game of Call to Power 2. Since the entire game extremely complex, we decided that an agent should only be able to invoke a subset of all the possible actions available in the game. It was decided to focus on the development of cities, units, and city improvements, and attaining victory through military conquest.

To that end, the source code for Call to Power 2 had to be modified to allow an outside program to control the actions of the computer player. The source code for Call to Power 2 was made open-source by Activision, and is maintained by an online community called Apolyton. [Apolyton, 2007] Ushhan's Gudevia used this source code to develop a basic API to control the computer player for his Master's thesis in 2006 [cite: Ushan's thesis]. That API allowed for the use of a few basic control commands, however it was not enough to allow for automatic control by a TIELT-enabled agent program. It was not able to provide any information as to whether a given command was successful or not, and what the results of the action were. For example, you could tell a Warrior to attack an enemy unit, but you could not determine if the combat was won or lost. Also, that API had only a very limited ability to communicate with TIELT. Other miscellaneous modifications were required as well, such as disabling modal dialogue windows that pop up to display information during the game, so that an agent could play through an entire session of the game without requiring human intervention. Therefore, this project required adding to the API to allow a TIELT-enabled agent to play a complete subset of the game with the functions available through the API, as well as enabling a complete integration with TIELT so that an agent could both query the status

of aspects of the game and sense any spontaneous changes to the state of the game (e.g.: having one of your units killed on an opponent's turn), and making a few miscellaneous optimizations to assist in the integration process.

The functions currently available in the API are as follows:

- `MoveArmyTo`: Instruct an army to move to a specified location
- `Settle`: Instruct a Settler to build a city on its current location
- `CityBuild`: Instructs one of your cities to build a specified type of unit
- `CityImprove`: Instructs a city to start building a city improvement of the specified type.
- `AttackEnemyPosWithArmy`: Instructs an army to attack an enemy-occupied location.
- `AttackCityPosWithArmy`: Instructs an army to attack a location where there is an enemy-controlled city.
- `ArmyToDefend`: Instruct an army to fortify the city at its current location.
- `StopDefending`: Instructs an army to stop fortifying its current location
- `FindEnemyUnit`: Searches for enemy units visible from one of your own units.
- `FindEnemyCity`: Searches for enemy cities visible from one of your own units.
- `FindUnexplored`: Searches for the closest unexplored territory an army can reach.
- `ArmyCanReach`: Determines whether or not there is a known path from an army to a location.
- `QueryCityBuildable`: Queries if a unit can build a city at its current location.
- `QueryMoveable`: Queries if a unit can move to a given location on the map.
- `QueryUnitBuildable`: Queries if a unit can be produced by a given city you control

- QueryImprovementBuildable: Queries if an improvement can be built in a given city you control.
- QueryGarrison: Queries if a unit can garrison its current location.
- QueryUngarrison: Queries if a unit can stop garrisoning its current location
- QueryUnitAttackable: Queries if a unit can attack an enemy unit on a given location
- QueryCityAttackable: Queries if a unit can attack an enemy city on a given location

The sensor messages added to inform TIELT of game state changes are as follows:

- An attack on an enemy city has succeeded
- An attack on an enemy unit has succeeded
- One of your units has been destroyed
- One of your cities has been destroyed
- A new city has been built
- A new unit has finished building
- Updates on the position of your units at the start of every turn
- Call To Power 2 has been started
- A game has started
- Your turn has started
- The game has ended
- Game has been lost
- Game has been won
- Call to Power 2 has been closed

Another thing worth mentioning is the abstraction provided by the API. Call to Power 2 uses a complex system of interrelated classes and objects to control different aspects of the game. It can become difficult to come to an understanding of this architecture in order to modify the game, both because of the complexity of the system architecture and because all the comments were removed from the source code by Activision. To simplify this problem, the API defines a few different classes in order to collect aspects of these internal objects into a single set of easy to understand objects.

The API provides four different classes, defined as follows:

### **API\_Player**

Description: This is a class that represents a one of the players in the game

Members:

- *int m\_iIndex*: this is a signed 32-bit integer which represents a single player, it refers to an index into a global array of players kept by the game
- *Player \*player*: this is a pointer into Call to Power 2's internal Player class

Methods:

- Constructors:
  - `API_Player( const int p_iIndex )`
  - `API_Player( const API_Player & p_pOther )`
- Destructor:
  - `~API_Player()`: destructor
- Accessors:
  - `int GetIndex()`

### **API\_Location**

Description: This is a class that represents a tile on the map

Members:

- *sint16 m\_iXCoord*: x-coordinate of this location

- *sint16 m\_iYCoord*: y-coordinate of this location
- *MapPoint point*: this is a reference to Call to Power 2's internal class for locations

Methods:

- Constructors:
  - `API_Location( const MapPoint p_pPoint )`
  - `API_Location( const sint16 p_iXCoord, const sint16 p_iYCoord )`
  - `API_Location( const API_Location & p_lOther )`
- Destructor:
  - `~API_Location()`
- Accessors:
  - `sint16 GetXCoord()`: access the x-coordinate member
  - `sint16 GetYCoord()`: access the y-coordinate member

## **API\_City**

Description: This is a class that represents a city on one of the tiles in the game

Members:

- *API\_Player m\_pPlayer*: reference to the player that owns this city
- *sint32 m\_iCityId*: integer identification for this city, this represents an index into an array of the player's cities

Methods:

- Constructors:
  - `API_City( const API_Player p_pPlayer, const sint32 p_iCityId );`
- Destructor:
  - `~API_City();`
- Accessors:
  - `API_Location GetLocation()`: access the location of the city
  - `sint32 GetIndex()`: access the index of the player who owns this city

## **API\_Army**

Description: This is a class that represents an army unit (military or non-military) on one of the tiles in the game

Members:

- *API\_Player m\_pPlayer*: reference to the player that owns this city
- *sint32 m\_iArmyId*: integer identification for this army, this represents an index into an array of the player's armies

Methods:

- Constructors:
  - `API_Army( const API_Player p_pPlayer, const sint32 p_iArmyId )`
- Destructor:
  - `~API_Army();`
- Accessors:
  - `API_Location GetLocation()`: access the location of the army
  - `sint32 GetIndex()`: access the index of the player who owns this army

It is worth noting that because of the abstraction of the game that this API provides, this project is not solely of use to researchers who wish to use TIELT. Since the API provides a simple C++ interface to control the actions of the computer player, any AI researcher could write an intelligent agent in C++ to interface with Call to Power 2 directly. It can even be of use to members of the Apolyton community who wish to make further modifications to the game's functionality since it simplifies ways of dealing with the details of the data structures used for controlling the game.

## **3.4 Code Reference**

The Call to Power 2 API is a constantly evolving project. The most current version of the API can always be found freely available at the InSyTe lab's website for the Call



to Power 2 – TIELT Integration project:

<http://www.cse.lehigh.edu/InSyTe/CTP2TieftIntegration/CTP2Integration.html>

## **4. The TIELT – Call To Power 2 Model**

### **4.1 What is the TIELT – Call to Power 2 model?**

In Section 2 we discussed the purpose of TIELT and explained the steps necessary to set up a domain for to model the rules of a game in TIELT. This section will explain specifically how Call to Power 2 was modeled in TIELT and some of the design decisions made for this purpose. Note that similar to the API, the Call to Power 2 model in TIELT also occasionally changes to fit the needs of the project. The TIELT specs can also be found online at the InSyTe lab website:

<http://www.cse.lehigh.edu/InSyTe/CTP2TielIntegration/CTP2Integration.html>

### **4.2 Environment Model**

As mentioned in Section 2, the TIELT Environment Model is where an abstract model of the game world is made. This includes the rules of the game, state variables, TIELT-specific classes, actions possible in the game world, and observations about changes to the game state. We will begin by discussing the TIELT-specific classes made since they will be referenced in the other parts of the game model.

Similar to how the API abstracts the internals of Call to Power 2 in four different classes, TIELT uses classes to abstract information about objects and state information in Call to Power 2. The first classes are those which correspond to the classes used by the API: API\_Player, API\_Location, API\_Army, and API\_City. Note that even though these classes share the same name as the classes in the API, they are not identical. The TIELT classes have member variables, but no associated methods. This is because these classes are used in order to wrap information passed through messages to the game into objects that are easier to conceive. These classes are defined as follows:

## **API\_Player**

Description: This is a class that represents a one of the players in the game

Members:

- *int m\_iIndex*: an integer which represents a single player

## **API\_Location**

Description: This is a class that represents a tile on the map

Members:

- *int m\_iXCoord*: x-coordinate of this location
- *int m\_iYCoord*: y-coordinate of this location

## **API\_City**

Description: This is a class that represents a city on one of the tiles in the game

Members:

- *API\_Player m\_pPlayer*: reference to the player that owns this city
- *sint32 m\_iCityId*: integer identification for this city, this represents an index into an array of the player's cities
- *int x\_pos*: the x-coordinate of this city
- *int y\_pos*: the y-coordinate of this city

## **API\_Army**

Description: This is a class that represents an army unit (military or non-military) on one of the tiles in the game

Members:

- *API\_Player m\_pPlayer*: reference to the player that owns this city
- *sint32 m\_iArmyId*: integer identification for this army, this represents an index into an array of the player's armies

In addition to these classes above, three more classes were added in order to represent sets of these objects. These classes are named `API_Location_List`, `API_Army_List`, and `API_City_List`, respectively. These classes were designed to function similarly to an `ArrayList` in Java, in that the lists could grow and shrink dynamically as needed as objects are added or removed from them. These objects are useful for representing collections of objects. For example, an `API_Army_List` is used for representing every unit a player controls in a single, easy to reference object. These classes are defined as follows:

### **API\_Location\_List**

Description: This is a class that represents a collection of `API_Location` objects

Members:

- `API_Location[] location_array`: an array of `API_Location` objects
- `int last_filled_index`: the last index in the array where an object has been placed

Methods:

- `void Insert( API_Location element )`: insert an `API_Location` object to the end of `location_array`
- `void Delete( int x_coord, int y_coord )`: delete the element in `location_array` whose x coordinate equals `x_coord` and whose y coordinate equals `y_coord`
- `int GetNumElements()`: access `last_filled_index`
- `int FindElement( int x_coord, int y_coord )`: return the index of the `API_Location` object in `location_array` whose x coordinate equals `x_coord` and whose y coordinate equals `y_coord`
- `void GrowArray()`: increase the size of the array
- `void ShrinkArray()`: decrease the size of the array

### **API\_City\_List**

Description: This is a class that represents a collection of `API_City` objects

Members:

- *API\_Location[] city\_array*: an array of *API\_Location* objects
- *int last\_filled\_index*: the last index in the array where an object has been placed

Methods:

- void *Insert( API\_City element )*: insert an *API\_City* object to the end of *location\_array*
- void *Delete( int cityid )*: delete the element in *city\_array* whose ID equals *cityid*
- int *GetNumElements()*: access *last\_filled\_index*
- int *FindElement( int x\_pos, int y\_pos )*: return the index of the *API\_City* object in *city\_array* whose x coordinate equals *x\_pos* and whose y coordinate equals *y\_pos*
- int *FindElement( int cityid )*: return the index of the *API\_City* object in *city\_array* whose ID equals *cityid*
- void *GrowArray()*: increase the size of the array
- void *ShrinkArray()*: decrease the size of the array

### **API\_Army\_List**

Description: This is a class that represents a collection of *API\_Army* objects

Members:

- *API\_Location[] army\_array*: an array of *API\_Army* objects
- *int last\_filled\_index*: the last index in the array where an object has been placed

Methods:

- void *Insert( API\_Army element )*: insert an *API\_Army* object to the end of *army\_array*
- void *Delete( int armyid )*: delete the element in *army\_array* whose ID equals *armyid*
- int *GetNumElements()*: access *last\_filled\_index*
- int *FindElement( int armyid )*: return the index of the *API\_Army* object in *army\_array* whose ID equals *armyid*
- void *GrowArray()*: increase the size of the array
- void *ShrinkArray()*: decrease the size of the array

With those classes defined, we can now look at the list of TIELT state variables used to model the current state of the world in Call to Power 2. As a design decision, we are not keeping an object to represent the number or locations of enemy army units. This is because enemy unit positions can change every turn, and due to the fog of war in Call to Power 2, it is extremely difficult to keep accurate information in such an object. Similarly, we do not keep track of the number of enemy units since the player cannot tell when an enemy unit that cannot be seen has died. The state variables used are as follows:

- `API_Location_List enemy_city_locations`: the list of locations for every enemy city encountered
- `API_Army_List my_army_locations`: the list of all armies you control
- `API_City_List my_city_locations`: the list of all cities you control
- `int curr_turn`: keeps track of how many turns have been taken
- `int my_player_id`: integer value of your player ID
- `int game_state`: integer representation of current game state: 0 = game lost, 1 = in progress, 2 = game won.
- `int num_enemy_cities`: the number of enemy cities you have found
- `int num_my_cities`: the number of cities you control
- `int num_my_armies`: the number of armies you control

Now we will explain the Action Models used defined for Call to Power 2. Recall that the Action Models represent the actions available for an agent to make in the environment. Also, keep in mind that only the actions required to play a subset of the

game have been implemented due to the complexity of the entire game. It should also be noted that due to the fact that some of the C++ API calls cannot give you an immediate result as to whether they succeeded or failed (for example, the AttackEnemyPosWithArmy function only returns whether it was possible to perform the attack) it was decided to implement a query-driven system for the Action Models. That is, if an agent wants to perform an action, it issues a query as to whether the action is possible. If the action is possible, the agent is notified that the action is possible and a command to perform that action is automatically issued by TIELT without requiring input from the agent. If the action is not possible, the agent is simply notified that the action is not possible. The only action that does not require a query is the action to end the current turn. The list of available TIELT Action Models follows:

### **Load a game**

Message Name:

Load

Description:

Loads a game

Parameters:

@name //name of game to be loaded

Relevant Phases:

Playing

Precondition:

*none*

Triggers message:

GAME\_LOAD

### **Save a game**

Message Name:

Save

Description:

Saves a game

Parameters:

@name //name of game to be loaded

Relevant Phases:

Playing

Precondition:

*none*

Triggers message:

GAME\_SAVE

### **End your turn**

Message Name:

EndTurn

Description:

Ends your current turn

Parameters:

*none*

Relevant Phases:

Playing

Precondition:

*none*

Triggers message:

END\_TURN

---

### **Query Attackable Enemy City**

Message Name:

QueryAttackCity

Description:

Query if a unit can attack an enemy city

Parameters:

@API\_Army p\_aArmy //army unit to attack with

@API\_Location p\_IDestination //location of city to attack

Relevant Phases:

Playing

Precondition:

-enemy city exists at that location

-your army unit is adjacent to that city

Triggers message:

QU\_ATTACK\_CITY

### **Query Attackable Enemy Unit**

Message Name:

QueryAttackUnit

Description:

Query if a unit can attack an enemy army

Parameters:

@API\_Army p\_aArmy //army unit to attack with

@API\_Location p\_IDestination //location of enemy unit to attack

Relevant Phases:

Playing

Precondition:



-your army unit is adjacent to that location

Triggers message:

QU\_ATTACK\_UNIT

### **Query Map**

Message Name:

QueryUnexploredMap

Description:

Query for an unexplored map square around a unit. Note that this will only provide one unexplored map square, and it is not guaranteed to find the nearest one.

Parameters:

@API\_Army p\_aArmy //army unit to do the exploration

Relevant Phases:

Playing

Precondition:

-the army unit to search with exists somewhere on the map

Triggers message:

QU\_UNEXPLORED\_MAP

### **Query Enemy Units**

QueryEnemyUnit

Description:

Query for enemy army units in visible range of a unit

Parameters:

@API\_Army p\_aArmy //army unit to do the exploration

@int p\_iVisionRange //Vision range of the unit (either 1 or two)

Relevant Phases:

Playing

Precondition:

-the army unit to search with exists somewhere on the map

Triggers message:

QU\_ENEMY\_UNIT

### **Query Enemy Cities**

QueryEnemyCity

Description:

Query for enemy cities in visible range of a unit

Parameters:

@API\_Army p\_aArmy //army unit to do the exploration

@int p\_iVisionRange //Vision range of the unit (either 1 or 2)

Relevant Phases:

Playing

Precondition:

-the army unit to search with exists somewhere on the map

Triggers message:

QU\_ENEMY\_CITY

### **Query Buildable Tile**

QueryCityBuildable

#### Description:

Query if a unit can build a city at its current location

#### Parameters:

@API\_Army p\_aArmy //army unit to build the city

#### Relevant Phases:

Playing

#### Preconditions:

-a unit with that army ID exists somewhere on the map

#### Triggers message:

QU\_CITY\_BUILDABLE

### **Query Moveable Tile**

QueryMoveable

#### Description:

Query if a given unit can move to a given tile (ex: Settlers can't move onto ocean, ships can't move onto land)

#### Parameters:

@API\_Army p\_aArmy //army unit to move

@API\_Location p\_lLocation //location to move that unit to

#### Relevant Phases:

Playing

#### Preconditions:

-that unit exists somewhere on the map

#### Triggers message:

QU\_MOVEABLE

### **Query Buildable Unit**

QueryUnitBuildable

#### Description:

Query if a unit can be built in a city (ie: you are far enough in the tech tree to build it)

#### Parameters:

@integer unit\_type //integer representation of unit you want to build

@API\_City p\_cCity //city you want to build that unit in

#### Relevant Phases:

Playing

#### Preconditions:

-the city to build the unit exists somewhere on the map

#### Triggers message:

QU\_UNIT\_BUILDABLE

### **Query Buildable City Improvement**

QueryImprovementBuildable

Description:

Query if a city improvement can be built in a specified city (ie: you are far enough in the tech tree to build it)

Parameters:

@integer improvement //integer representation of improvement you want to build  
@API\_City p\_cCity //city you want to build that improvement in

Relevant Phases:

Playing

Preconditions:

-the city to build the improvement exists somewhere on the map

Triggers message:

QU\_IMPROVEMENT\_BUILDABLE

**Query if a unit can defend a City**

Message Name:

QueryGarrison

Description:

Queries if a unit can garrison its current location

Parameters:

@API\_Army p\_aArmy //army unit to defend with

Relevant Phases:

Playing

Precondition:

-a city exists at the destination  
-unit is on the city it will garrison

Triggers message:

QU\_GARRISON

**Query if a unit can stop defending a city**

Message Name:

QueryUngarrison

Description:

Queries if a unit can stop garrisoning its current location

Parameters:

@API\_Army p\_aArmy //unit to be ungarrisoned

Relevant Phases:

Playing

Precondition:

-an army unit is garrisoning a city

Triggers message:

QU\_UNGARRISON

**Table 1: Call to Power 2 Action Models**

Next, we will define the Observation Models created for Call to Power 2.

Observation Models represent our ability to sense changes in the game world. When an

Observation Model is triggered, a change may be made to the TIELT state variables in order to model the new state of the game world. It is worth mentioning that it was decided to make state information updates “push-driven” for this project. This means that the game state is only updated when information about a state change is sent from Call to Power 2 to TIELT. In other words, TIELT never assumes its action changes the current state. It tells the game engine what it wants to do and then waits for information on what happened as a result of the desired action(s). Hence, for the Observation Models, we will also list any corresponding state changes that they cause. The Observation Models are as follows:

### **Enemy City Destroyed**

Message Name:

EnemyCityDestroyed

Description:

One of your units destroyed an enemy city

Parameters:

|             |  |
|-------------|--|
| @int armyID | //ID of army that attacked the city    |
| @int x_pos  | //x-position of city that was attacked |
| @int y_pos  | //y-position of city that was attacked |

Conditions:

*none*

Triggered By:

UPD\_ATTACK\_CITY returning success

### **Enemy Unit Destroyed**

Message Name:

EnemyUnitDestroyed

Description:

One of your units destroyed an enemy unit

Parameters:

|             |   |
|-------------|---|
| @int armyID | //ID of army that attacked the unit               |
| @int x_pos  | //x-position of army that attacked the enemy unit |
| @int y_pos  | //y-position of army that attacked the enemy unit |

**Note: This does not return the x-y position of the unit that was killed**

Conditions:

*none*

Triggered By:

UPD\_ATTACK\_UNIT returning success

### **My Unit Destroyed**

Message Name:

MyUnitDestroyed

Description:

One of your units was destroyed

Parameters:

|             |   |
|-------------|---|
| @int armyID | //ID of army that was destroyed             |
| @int x_pos  | //x-position of the unit that was destroyed |
| @int y_pos  | //y-position of the unit that was destroyed |

Conditions:

*none*

Triggered By:

DESTROYED\_UNIT

### **My City Destroyed**

Message Name:

MyCityDestroyed

Description:

One of your cities was destroyed

Parameters:

|             |   |
|-------------|---|
| @int cityID | //ID of city that was destroyed             |
| @int x_pos  | //x-position of the unit that was destroyed |
| @int y_pos  | //y-position of the unit that was destroyed |

Conditions:

*none*

Triggered By:

DESTROYED\_CITY

### **New City Created**

Message Name:

CitySettled

Description:

A new city has been settled

Parameters:

|             |                              |
|-------------|------------------------------|
| @int cityID | //ID of the newly built city |
| @int x_pos  | //x position of city built   |
| @int y_pos  | //y position of city built   |

Conditions:

*none*

Triggered By:

UPD\_SETTLED\_CITY\_ID

### **New Unit Finished Building**

Message Name:

## NewUnitComplete

### Description:

A new unit has finished building

### Parameters:

|             |  |
|-------------|--|
| @int armyID | //ID of army unit being produced             |
| @int x_pos  | //x-position of the city making the new unit |
| @int y_pos  | //y-position of the city making the new unit |

### Conditions:

*none*

### Triggered By:

NEW\_UNIT\_COMPLETED

## Unexplored Map Tile

### Message Name:

UnexploredTile

### Description:

Query response with x-y position of an unexplored tile by a unit. Note that this is not guaranteed to be the closest unexplored tile with respect to the unit.

### Parameters:

|               |  |
|---------------|--|
| @bool success | //set true if there is an unexplored tile around that unit |
| @int armyID   | //ID of army that is performing the search                 |
| @int x_pos    | //x-position of the tile returned                          |
| @int y_pos    | //y-position of the tile returned                          |

### Conditions:

*none*

### Triggered By:

UPD\_UNEXPLORED\_MAP

## Enemy Units Nearby

### Message Name:

NearEnemyUnit

### Description:

Query response with x-y positions of an enemy unit near a unit you control

Note: this is triggered once for each enemy unit near the unit you are querying with respect to

### Parameters:

|               |  |
|---------------|--|
| @bool success | //set true if there are any enemy units in visual range of your unit |
| @int x_pos    | //x position of the enemy unit found                                 |
| @int y_pos    | //y position of the enemy unit found                                 |

### Conditions:

*none*

### Triggered By:

UPD\_ENEMY\_UNIT

## Enemy Cities Nearby

Message Name:

NearEnemyCity

Description:

Query response with x-y positions of an enemy city near a unit you control

Note: this is called once for each unit near the unit you are querying with respect to

Parameters:

@bool success //set true if there are any enemy units in visual range of your unit

@int x\_pos //x position of the enemy city found

@int y\_pos //y position of the enemy city found

Conditions:

*none*

Triggered By:

UPD\_ENEMY\_CITY

**Unit Position Update**

Message Name:

UpdatePosition

Description:

Informs you of new position for unit with ID of armyID

Parameters:

@int armyID //ID of army unit that has been moved

@int old\_x //old x position of the unit

@int old\_y //old y position of the unit

@int new\_x //new x position of the unit

@int new\_y //new y position of the unit

Conditions:

*none*

Triggered By:

UPD\_ARMY\_XY

**City Can Be Attacked**

Message Name:

CityAttackable

Description:

Query response whether a city can be attacked by a given unit

Parameters:

@bool success //set true if the city can be attacked

@int armyID //ID of the unit that will attack

@int x\_pos //x-location of the city to be attacked

@int y\_pos //y-location of the city to be attacked

Conditions:

*none*

Triggered By:

UPD\_CITY\_ATTACKABLE

### **Unit Can Be Attacked**

Message Name:

UnitAttackable

Description:

Query response whether a unit can be attacked by a given unit

Parameters:

|               |   |
|---------------|---|
| @bool success | //set true if the unit can be attacked  |
| @int armyID   | //ID of the unit that is attacking      |
| @int x_pos    | //x-location of the unit to be attacked |
| @int y_pos    | //y-location of the unit to be attacked |

Conditions:

*none*

Triggered By:

UPD\_UNIT\_ATTACKABLE

### **City Can Be Built**

Message Name:

CityBuildable

Description:

Query response whether a city can be built on a given tile

Parameters:

|               |  |
|---------------|--|
| @bool success | //set true if there is an a city can be built on the x,y position passed in from QU_CITY_BUILDABLE message |
| @int armyID   | //ID of the unit that will build the city  |
| @int x_pos    | //x-location where city will be built  |
| @int y_pos    | //y-location where city will be built  |

Conditions:

*none*

Triggered By:

UPD\_CITY\_BUILDABLE

### **Unit Can Be Moved**

Message Name:

UnitMoveable

Description:

Query response whether a unit can move to a given tile

Parameters:

|               |  |
|---------------|--|
| @bool success | //set true if the unit in question can move to the tile passed in from the QU_MOVEABLE message |
| @int armyID   | //ID of the unit that will move  |
| @int x_pos    | //x-location where city will be built  |
| @int y_pos    | //y-location where city will be built  |

Conditions:

*none*

Triggered By:

UPD\_MOVEABLE



### **Unit Can Be Built**

Message Name:

UnitBuildable

Description:

Query response whether a unit can be built

Parameters:

@bool success //set true if the unit type in question can be built  
@int cityID //city the unit will be built in  
@int unit\_type //type of unit being built

Conditions:

*none*

State Changes:

*none*

Triggered By:

UPD\_UNIT\_BUILDDABLE

### **City Improvement Can Be Built**

Message Name:

ImprovementBuildable

Description:

Query response whether a city improvement can be built

Parameters:

@bool success //set true if the improvement type in question can be built  
@int cityID //city the improvement will be built in  
@int improvement\_type //type of city improvement to build

Conditions:

*none*

Triggered By:

UPD\_IMPROVEMENT\_BUILDDABLE

### **Unit Is Garrisoning**

Message Name:

Garrisoning

Description:

Indicate whether a unit is garrisoning

Parameters:

@bool success //set true if the unit is garrisoning a city  
@int armyID //ID of unit that is garrisoning  
@int x\_pos //x-position of the unit  
@int y\_pos //y-position of the unit

Conditions:

*none*

Triggered By:

UPD\_GARRISON

### **Unit Has Stopped Garrisoning**

Message Name:

StoppedGarrisoning

Description:

Indicate whether a unit has stopped garrisoning

Parameters:

|               |  |
|---------------|--|
| @bool success | //set true if the unit is garrisoning a city |
| @int armyID   | //ID of unit that is garrisoning             |
| @int x_pos    | //x-position of the unit                     |
| @int y_pos    | //y-position of the unit                     |

Conditions:

*none*

Triggered By:

UPD\_UNGARRISON

### **A Turn Has Ended**

Message Name:

TurnOver

Description:

Indicates that any turn has ended

Parameters:

*none*

Conditions:

*none*

Triggered By:

GAME\_TICK

### **Your Turn Has Started**

Message Name:

MyTurn

Description:

Indicates that your turn has begun

Parameters:

*none*

Conditions:

*none*

Triggered By:

GAME\_MYTURN

### **Your Player ID**

Message Name:

MyID

Description:

Indicates what your player ID is

Parameters:

|               |                  |
|---------------|------------------|
| @int playerID | //your player ID |
|---------------|------------------|

Conditions:

*none*

Triggered By:

GAME\_MYID

### **Total Number of Players**

Message Name:

NumPlayers

Description:

Indicates how many players are in the current game

Parameters:

@int num\_players //total number of players

Conditions:

*none*

Triggered By:

GAME\_NUM\_PLAYERS

**Note: this is supported in TIELT, but the message is not yet implemented in CTP2**

### **Program Begins**

Message Name:

Connected

Description:

Indicates start of CTP2 program

Parameters:

@int num\_players //total number of players

Conditions:

*none*

Triggered By:

CTP2\_START

### **Game Has Started**

Message Name:

GameStarted

Description:

Indicates start of an individual game

Parameters:

*none*

Conditions:

*none*

Triggered By:

GAME\_START

### **Lost Game**

Message Name:

Lose

Description:

Indicates a game loss

Parameters:

*none*

Conditions:

*none*

Triggered By:

GAME\_LOSE

### **Won Game**

Message Name:

Win

Description:

Indicates a game win

Parameters:

*none*

Conditions:

*none*

Triggered By:

GAME\_WIN

### **Saved Game**

Message Name:

GameSaved

Description:

Indicates success or failure of attempt to save a game

Parameters:

@bool success            //set to true if game saved successfully, otherwise false

Conditions:

*none*

Triggered By:

UPD\_SAVE

### **Loaded Game**

Message Name:

GameLoaded

Description:

Indicates success or failure of attempt to load a game

Parameters:

@bool success            //set to true if game saved successfully, otherwise false

Conditions:

*none*

Triggered By:

UPD\_LOAD

### **Game Finished**

Message Name:

**GameOver**  
Description:  
Indicates end of an individual game  
Parameters:  
*none*  
Conditions:  
*none*  
Triggered By:  
GAME\_DONE

**Program Closed**  
Message Name:  
Disconnected  
Description:  
Indicates end of CTP2 program  
Parameters:  
*none*  
Conditions:  
*none*  
Triggered By:  
CTP2\_END

**Table 2: Call to Power 2 Observation Models**

### **4.3 Simulator Interface**

The Simulator Interface model defines the messages which TIELT uses in order to communicate with the game engine. The Simulator Interface is broken down into outgoing messages from TIELT to Call to Power 2, and incoming messages from Call to Power 2 to TIELT. We will start with the outgoing messages. Note that these messages will also make use of instantiated objects of the TIELT-specific classes that we defined in the Environment Model. Also, there will be some messages defined here that are used exclusively for handshaking with Call to Power 2 when initiating the TIELT-Call to Power 2 connection before starting an experiment. The outgoing messages used for this project are as follows:

**Attack Enemy City**  
Message Name:  
ACT\_ATTACK\_CITY

Description:

Instructs a unit to attack an enemy city

Message Arguments:

@API\_Army p\_aArmy //army unit to attack with  
@API\_Location p\_IDestination //location of city to attack

Triggered By:

Incoming Message: UPD\_CITY\_ATTACKABLE

API Functions this will call:

AttackCityPosWithArmy( API\_Army p\_aArmy, const API\_Location p\_IDestination )

Replied to with:

UPD\_ATTACK\_CITY if attack was won

DESTROYED\_UNIT if attack was lost

**Attack Enemy Unit**

ACT\_ATTACK\_UNIT

Description:

Instructs a unit to attack an enemy army

Message Arguments:

@API\_Army p\_aArmy //army unit to attack with  
@API\_Location p\_IDestination //location of enemy unit to attack

Triggered By:

Incoming Message: UPD\_UNIT\_ATTACKABLE

API Functions this will call:

AttackEnemyPosWithArmy( API\_Army p\_aArmy, const API\_Location p\_IDestination )

Replied to with:

UPD\_ATTACK\_UNIT if attack was won

DESTROYED\_UNIT if attack was lost

**Build City**

Message Name:

ACT\_SETTLE

Description:

Instructs a Settler to move to a point on the map and build a city there

Message Arguments:

@API\_Army p\_aArmy //Settler to build the new city

Triggered By:

Incoming message: UPD\_CITY\_BUILDABLE

API Functions this will call:

Settle( API\_Army p\_aArmy )

Replied to with:

UPD\_SETTLED\_CITY\_ID

**Defend a City**

Message Name:

ACT\_GARRISON

Description:

Instructs a unit to garrison at its current location

Message Arguments:

@API\_Army p\_aArmy //army unit to defend with

Triggered By:

Incoming Message: UPD\_GARRISON

API Functions this will call:

ArmyToDefend( API\_Army p\_aArmy );

Replied to with:

No need for a response since garrisoning is a deterministic action. UPD\_GARRISON will trigger the Garrisoning observation model.

### **Move a Unit**

Message Name:

ACT\_MOVE

Description:

Instruct a unit to move to a specified location

Message Arguments:

@API\_Army p\_aArmy //unit to be moved

@API\_Location p\_IDestination //location to move that unit to

Triggered By:

Incoming Message: UPD\_MOVEABLE

API Functions this will call:

MoveArmyTo( API\_Army p\_aArmy, const API\_Location p\_IDestination )

Replied to with:

No immediate reply, but UPD\_ARMY\_XY will be sent at the start of the next turn to indicate where the moved army is

### **Create a Unit**

Message Name:

ACT\_PRODUCE

Description:

Create a unit in a specified city

Message Arguments:

@API\_City p\_cCity //city to create the unit

@int p\_iUnitType //integer representation of unit to create

Triggered By:

Incoming Message: UPD\_UNIT\_BUILDABLE

API Functions this will call:

CityBuild( API\_City p\_cCity, const API\_UnitType p\_iUnitType )

Replied to with:

No immediate reply because the unit is placed in the build queue and takes some number of turns to complete. When the unit is finished NEW\_UNIT\_COMPLETED will be sent.

### **Improve a City**

Message Name:

ACT\_IMPROVE

Description:

Build a city improvement in a specified city

Message Arguments:

@API\_City p\_cCity //city to build the improvement

@int p\_iImproveType //integer representation of improvement type to create

Triggered By:

Incoming Message: UPD\_IMPROVEMENT\_BUILDABLE

API Functions this will call:

CityImprove( API\_City p\_cCity, const API\_CityImprovementType p\_iImproveType )

Replied to with:

No reply because the improvement gets placed in the build queue. There is no notification when the improvement is completed because city improvements are not tracked in the game state.

**Stop a Garrison**

ACT\_UNGARRISON

Description:

Stop a unit from garrisoning a city

Message Arguments:

@API\_Army p\_aArmy //unit to stop garrisoning

Triggered By:

Incoming Message: UPD\_UNGARRISON

API Functions this will call:

StopDefending( API\_Army p\_aArmy )

Replied to with:

No need for a response since ungarrisoning is a deterministic action.

UPD\_UNGARRISON will trigger the StoppedGarrisoning observation model.

## Game Messages

**Load a game**

GAME\_LOAD

Description:

Loads a game

Message Arguments:

@name //name of game to be loaded

Triggered By:

Action Model: Load

API Functions this will call:

LoadGame

Replied to with:

UPD\_LOAD

**Save a game**

GAME\_SAVE

Description:



Saves the current game

Message Arguments:

@name //name of game to be saved

Triggered By:

Action Model: Save

API Functions this will call:

SaveGame

Replied to with:

UPD\_SAVE

### **End the Turn**

END\_TURN

Description:

Ends your current turn

Message Arguments:

*none*

Triggered By:

Action Model: EndTurn

API Functions this will call:

None

Replied to with:

*None*

### **Acknowledge game start**

HELLO

Description:

Acknowledges program starting

Message Arguments:

*none*

Triggered By:

Incoming Message: CTP2\_START

API Functions this will call:

None

Replied to with:

GAME\_START

## **Query Messages**

### **Query Attackable Enemy City**

Message Name:

QU\_ATTACK\_CITY

Description:

Query whether your unit can attack an enemy city

Message Arguments:

@API\_Army p\_aArmy //army unit to do the exploration

@API\_Location p\_IDestination //location of city to attack

Triggered By:

Action Model: QueryAttackCity

API Functions this will call:

QueryCityAttackable( API\_Army p\_aArmy, API\_Location p\_IDestination );

Replied to with:

UPD\_CITY\_ATTACKABLE

**Query Attackable Enemy Unit**

Message Name:

QU\_ATTACK\_UNIT

Description:

Query whether your unit can attack an enemy unit

Message Arguments:

@API\_Army p\_aArmy //army unit to do the exploration

@API\_Location p\_IDestination //location of city to attack

Triggered By:

Action Model: QueryAttackUnit

API Functions this will call:

QueryUnitAttackable( API\_Army p\_aArmy, API\_Location p\_IDestination );

Replied to with:

UPD\_UNIT\_ATTACKABLE

**Query Map**

Message Name:

QU\_UNEXPLORED\_MAP

Description:

Query for an unexplored map square around a unit. Note that this will only provide one unexplored map square, and it is not guaranteed to find the nearest one.

Message Arguments:

@API\_Army p\_aArmy //army unit to do the exploration

Triggered By:

Action Model: QueryUnexploredMap

API Functions this will call:

FindUnexplored( API\_Army p\_aArmy, API\_Location & p\_IUnexplored )

Replied to with:

UPD\_UNEXPLORED\_MAP

**Query Enemy Units**

QU\_ENEMY\_UNIT

Description:

Query for enemy army units in visible range of a unit

Message Arguments:

@API\_Army p\_aArmy //army unit to do the exploration

@int p\_iVisionRange //Vision range of the unit (either 1 or two)

Triggered By:

Action Model: QueryEnemyUnit

API Functions this will call:

FindEnemyUnit( API\_Army p\_aArmy, const int p\_iVisionRange, DynamicArray<Unit>  
\* p\_pEnemyList )

Replied to with:

UPD\_ENEMY\_UNIT

### **Query Enemy Cities**

QU\_ENEMY\_CITY

Description:

Query for enemy cities in visible range of a unit

Message Arguments:

@API\_Army p\_aArmy //army unit to do the exploration

@int p\_iVisionRange //Vision range of the unit (either 1 or two)

Triggered By:

Action Model: QueryEnemyCity

API Functions this will call:

FindEnemyCity( API\_Army p\_aArmy, const int p\_iVisionRange, DynamicArray<Unit>  
\* p\_pCityList )

Replied to with:

UPD\_ENEMY\_CITY

### **Query Buildable Tile**

QU\_CITY\_BUILDDABLE

Description:

Query if a unit can build a city at its current location

Message Arguments:

@API\_Army p\_aArmy //army unit to build the city

Triggered By:

Action Model: QueryCityBuildable

API Functions this will call:

CityBuild( API\_City p\_cCity, const API\_UnitType p\_iUnitType );

Replied to with:

UPD\_CITY\_BUILDDABLE

### **Query Moveable Tile**

QU\_MOVEABLE

Description:

Query if a given unit can move to a given tile (ex: Settlers can't move onto ocean, ships  
can't move onto land)

Message Arguments:

@API\_Army p\_aArmy //army unit to move

@API\_Location p\_iLocation //location to move that unit to

Triggered By:

Action Model: QueryMoveable

API Functions this will call:

QueryMoveable( API\_Army p\_aArmy, API\_Location p\_iLocation );

Replied to with:  
UPD\_MOVEABLE

**Query Buildable Unit**  
QU\_UNIT\_BUILDDABLE

Description:

Query if a given unit can be built in a city (ie: you are far enough in the tech tree to build it)

Message Arguments:

    @integer    unit\_type    //integer representation of unit you want to build  
    @API\_City  p\_cCity      //city you want to build that unit in

Triggered By:

Action Model: QueryUnitBuildable

API Functions this will call:

QueryUnitBuildable( int unit\_type, API\_City p\_cCity );

Replied to with:

UPD\_UNIT\_BUILDDABLE

**Query Buildable City Improvement**  
QU\_IMPROVEMENT\_BUILDDABLE

Description:

Query if a city improvement can be built in a specified city (ie: you are far enough in the tech tree to build it)

Message Arguments:

    @integer    improvement  //integer representation of the improvement you  
    want to build  
    @API\_City  p\_cCity      //city you want to build the improvement in

Triggered By:

Action Model: QueryImprovementBuildable

API Functions this will call:

QueryImprovementBuildable( int improvement\_type, API\_City p\_cCity );

Replied to with:

UPD\_IMPROVEMENT\_BUILDDABLE

**Query Garrisoning Unit**  
QU\_GARRISON

Description:

Query whether a given unit can garrison its current location

Message Arguments:

    @API\_Army  p\_aArmy      //army you are querying about

Triggered By:

Action Model: QueryGarrison

API Functions this will call:

QueryGarrison( API\_Army p\_aArmy );

Replied to with:

UPD\_GARRISON

### **Query Stopping a Garrison**

QU\_UNGARRISON

Description:

Query whether a given unit can stop garrisoning its current location

Message Arguments:

@API\_Army p\_aArmy //army you are querying about

Triggered By:

Action Model: QueryUngarrison

API Functions this will call:

QueryUngarrison( API\_Army p\_aArmy )

Replied to with:

UPD\_UNGARRISON

**Table 3: Call to Power 2 Outgoing Messages**

Peer to the outgoing messages are the incoming messages. These messages are sent either in response to a query from TIELT or in response to a change in the game state in Call to Power 2. Similar to the outgoing messages, there will be some incoming messages listed that are used for handshaking when the program is started. The incoming messages are as follows:

### **Update: Attack Enemy City**

Message Name:

UPD\_ATTACK\_CITY

Description:

Update to indicate success or failure of a unit you control (with ID of armyID) attacking city at location (x,y)

Message Arguments:

@bool success //set true if unit destroyed city, false if it died in the process  
@int armyID //ID of army that attacked the city  
@int x\_pos //x-position of city that was attacked  
@int y\_pos //y-position of city that was attacked

Triggered By:

ACT\_ATTACK\_CITY

State update:

If (success)

Remove enemy city from enemy\_city\_locations array  
num\_enemy\_cities--  
Call: EnemyCityDestroyed observation model

### **Update: Attack Enemy Army**

Message Name:

UPD\_ATTACK\_UNIT

Description:

Update to indicate success or failure of a unit you control (with ID of armyID ) attacking an enemy unit at location (x,y)

Message Arguments:

@bool success //set true if your unit won, false if it died in the process  
@int armyID //ID of army that attacked the city  
@int x\_pos //x-position of unit that you attacked with  
@int y\_pos //y-position of unit that you attacked with

**Note: this does not report the x-y position of the unit that was destroyed**

Triggered By:

ACT\_ATTACK\_UNIT

State update:

If (success)

Call: EnemyUnitDestroyed observation model

**Update: Settled City ID**

Message Name:

UPD\_SETTLED\_CITY\_ID

Description:

Update to indicate the ID of the city built by the settler and location passed in by the ACT\_SETTLE message

Message Arguments:

@int cityID //ID of the newly built city  
@int x\_pos //x position of city built  
@int y\_pos //y position of city built

Triggered By:

ACT\_SETTLE

State update:

Add city to my\_city\_locations array

num\_my\_cities++

Call: CitySettled observation model

**Update: New Unit Finished Building**

Message Name:

NEW\_UNIT\_COMPLETED

Description:

Update to indicate information about a new unit that has finished building

Message Arguments:

@int armyID //ID of army unit being produced  
@int x\_pos //x-position of the new unit  
@int y\_pos //y-position of the new unit  
@int type //the integer representation of the type of unit made

Triggered By:

-a new unit being created

State update:

Add this new unit to my\_army\_locations array

Call: NewUnitBuilding observation model

**Update: Query Unexplored Map**

Message Name:

UPD\_UNEXPLORED\_MAP

Description:

Update to indicate an unexplored tile near one of your units

Message Arguments:

@bool success //set true if there is an unexplored tile around that unit  
@int armyID //ID of army that is performing the search  
@int x\_pos //x-position of the tile returned  
@int y\_pos //y-position of the tile returned

Triggered By:

QU\_UNEXPLORED\_MAP

State update:

Call: UnexploredTile observation model

**Update: Query Enemy Unit**

Message Name:

UPD\_ENEMY\_UNIT

Description:

Update to indicate the existence of enemy units in visual range of a specific army unit you control

Message Arguments:

@bool success //set true if there are any enemy units in visual range of your unit  
@Array[] int positions //array of x and y positions (respectively) for units found

Triggered By:

QU\_ENEMY\_UNIT

-also triggered at the start of every turn if there are enemy units nearby your units

State update:

if (success)

Call: NearEnemyUnit observation model

**Update: Query Enemy City**

Message Name:

UPD\_ENEMY\_CITY

Description:

Update to indicate the existence of enemy cities in visual range of a specific army unit you control

Message Arguments:

@bool success //set true if there are any enemy cities in visual range of your unit

@Array[] int positions //array of x and y positions (respectively) for cities found

Triggered By:

QU\_ENEMY\_CITY

-also triggered at the start of every turn if there are enemy cities nearby

State update:

if (success)

Add these locations to enemy\_city\_locations array

num\_enemy\_cities = num\_enemy\_cities + length of array

Call: NearEnemyCity observation model

**Update: Army X,Y position**

Message Name:

UPD\_ARMY\_XY

Description:

Update to indicate an army's x,y location

Message Arguments:

@int armyID //ID of army unit that has been moved

@int x\_pos //x position of the unit

@int y\_pos //y position of the unit

@int type //the integer representation of the type of unit moved

Triggered By:

Start of your turn

State update:

Set army x and y position in my\_army\_locations array

Call: UpdatePosition observation model

**Update: Buildable Tile**

Message Name:

UPD\_CITY\_BUILDABLE

Description:

Update to indicate whether a city can be built on a certain tile

Message Arguments:

@bool success //set true if there is an a city can be built on the x,y position passed in from QU\_CITY\_BUILDABLE message

@int armyID //ID of the unit that will build the city

@int x\_pos //x-location where city will be built

@int y\_pos //y-location where city will be built

Triggered By:

QU\_CITY\_BUILDABLE

State update:

Call: CityBuildable observation model

If (success)

Call: ACT\_SETTLE outgoing message to build the city there

**Update: Moveable Tile**



Message Name:

UPD\_MOVEABLE

Description:

Update to indicate whether a unit can be moved to a certain tile

Message Arguments:

|               |  |
|---------------|--|
| @bool success | //set true if the unit in question can move to the tile passed in from the QU_MOVEABLE message |
| @int armyID   | //ID of the unit that will move  |
| @int x_pos    | //x-location where the unit is   |
| @int y_pos    | //y-location where the unit is   |
| @int x_dest   | //x-location the unit will move to   |
| @int y_dest   | //y-location the unit will move to   |

Triggered By:

QU\_MOVEABLE

State update:

Call: UnitMoveable observation model

If (success)

Call: ACT\_MOVE outgoing message to move the unit there

**Update: Unit Buildable**

Message Name:

UPD\_UNIT\_BUILDABLE

Description:

Update to indicate whether a unit can be built (ie: you are far enough in the tech tree to build it)

Message Arguments:

|                |  |
|----------------|--|
| @bool success  | //set true if the unit type in question can be built |
| @int cityID    | //city unit will be built in                         |
| @int unit_type | //type of unit to build                              |

Triggered By:

QU\_UNIT\_BUILDABLE

State update:

Call: UnitBuildable observation model

If (success)

Call: ACT\_PRODUCE outgoing message to make the unit

**Update: City Improvement Buildable**

Message Name:

UPD\_IMPROVEMENT\_BUILDABLE

Description:

Update to indicate whether a city improvement can be built (ie: you are far enough in the tech tree to build it)

Message Arguments:

|                       |   |
|-----------------------|---|
| @bool success         | //set true if the improvement type in question can be built |
| @int cityID           | //ID of city to build the improvement                       |
| @int improvement_type | //type of improvement to build                              |

Triggered By:

QU\_IMPROVEMENT\_BUILDABLE

State update:

Call: ImprovementBuildable observation model

If (success)

Call: ACT\_IMPROVE outgoing message

**Update: Unit Garrisoning**

Message Name:

UPD\_GARRISON

Description:

Update to indicate whether a given unit can garrison its current location

Message Arguments:

@bool success //set true if the unit is garrisoning a city

@int armyID //ID of the unit that is now garrisoning

Triggered By:

QU\_GARRISON

State update:

-change garrisoning Boolean variable for armyID in my\_army\_locations array to true

Call: Garrisoning observation model

**Update: Unit Stopped Garrisoning**

Message Name:

UPD\_UNGARRISON

Description:

Update to indicate whether a given unit can stop garrisoning its current location

Message Arguments:

@bool success //set true if the unit could stop garrisoning

@int armyID //ID of the unit that is now ungarrisoned

Triggered By:

QU\_UNGARRISON

State update:

-change garrisoning Boolean variable for armyID in my\_army\_locations array to false

Call: StoppedGarrisoning observation model

**Update: City Attackable**

Message Name:

UPD\_CITY\_ATTACKABLE

Description:

Update to indicate whether a given unit can attack a given city

Message Arguments:

@bool success //set true if you can attack the city

@int armyID //ID of the unit that will attack

@int x\_pos //x-position of city that will be attacked

@int y\_pos //y-position of city that will be attacked

Triggered By:

QU\_ATTACK\_CITY

State update:

Call: CityAttackable observation model

### **Update: Unit Attackable**

Message Name:

UPD\_UNIT\_ATTACKABLE

Description:

Update to indicate whether a given unit you control can attack an enemy unit

Message Arguments:

|               |  |
|---------------|--|
| @bool success | //set true if you can attack the enemy unit    |
| @int armyID   | //ID of the unit that will attack              |
| @int x_pos    | //x-position of army that you will attack with |
| @int y_pos    | //y-position of army that you will attack with |

Triggered By:

QU\_ATTACK\_UNIT

State update:

Call: UnitAttackable observation model

### **City Destroyed**

Message Name:

DESTROYED\_CITY

Description:

Update to indicate that one of your cities has been destroyed

Message Arguments:

|             |                        |
|-------------|------------------------|
| @int cityID | //ID of city destroyed |
|-------------|------------------------|

Triggered By:

-one of your cities being destroyed

State update:

Remove enemy city from enemy\_city\_locations array

num\_enemy\_cities--

Call: MyCityDestroyed observation model

### **Unit Destroyed**

Message Name:

DESTROYED\_UNIT

Description:

Update to indicate that one of your units has been destroyed

Message Arguments:

|             |                        |
|-------------|------------------------|
| @int armyID | //ID of army destroyed |
|-------------|------------------------|

Triggered By:

-one of your units being destroyed

State update:

Remove unit with ID of armyID from my\_army\_locations array

num\_my\_armies--

Call: MyUnitDestroyed observation model

---

### **Game Turn passed**

Message Name:

GAME\_TICK

Description:

Update to indicate the end of *any* turn

Message Arguments:

*none*

Triggered By:

-Any turn ending

State update:

curr\_turn++

Call: TurnOver observation model

### **Your Turn begins**

Message Name:

GAME\_MYTURN

Description:

Update to indicate that it is now your turn

Message Arguments:

*none*

Triggered By:

-Enemy turns ending

State update:

Call: MyTurn observation model

### **Player ID**

Message Name:

GAME\_MY\_ID

Description:

Update to tell you what your player ID is

Message Arguments:

@int playerID //your player ID

Triggered By:

-game starting

State update:

my\_player\_id = playerID

Call: MyID observation model

### **Number of Players**

Message Name:

GAME\_NUM\_PLAYERS

Description:

Update to tell you how many players are in the current game

Message Arguments:

@int num\_players //total number of players

Triggered By:

- 1) game starting
- 2) enemy player has been defeated

State update:

num\_enemy\_players = num\_players – 1;

Call: NumPlayers observation model

**Note: Although it is supported in TIELT, CTP2 does not currently send this message**

**Program Start**

Message Name:

CTP2\_START

Description:

Message to indicate start of CTP2 program

Message Arguments:

*none*

Triggered By:

- program starting

State update:

Call: Connected observation model

Call: HELLO outgoing message

**Game Start**

Message Name:

GAME\_START

Description:

Message to indicate start of game

Message Arguments:

*none*

Triggered By:

- game starting

State update:

Call: GameStarted observation model

**Game Lost**

Message Name:

GAME\_LOSE

Description:

Update to indicate a loss

Message Arguments:

*none*

Triggered By:

-losing the game

State update:

Call: Lose observation model

### **Game Won**

Message Name:

GAME\_WIN

Description:

Update to indicate a win

Message Arguments:

*none*

Triggered By:

-winning the game

State update:

Call: Win observation model

### **Game Finished**

Message Name:

GAME\_DONE

Description:

Message to indicate completion of game

Message Arguments:

*none*

Triggered By:

-game complete

State update:

Call: GameOver observation model

### **Program End**

Message Name:

CTP2\_END

Description:

Message to indicate end of CTP2 program

Message Arguments:

*none*

Triggered By:

- program ending

State update:

Call: Disconnected observation model

### **Game Loaded**

Message Name:

UPD\_LOAD

Description:

Indicates success or failure of loading a game

Message Arguments:

@bool success //set true if game was loaded, false otherwise

Triggered By:

GAME\_LOAD

State update:

Call: GameLoaded observation model

### **Game Saved**

Message Name:

UPD\_SAVE

Description:

Indicates success or failure of saving a game

Message Arguments:

@bool success //set true if game was saved, false otherwise

Triggered By:

GAME\_SAVE

State update:

Call: GameSaved observation model

**Table 4: Call to Power 2 Incoming Messages**

## **4.4 Communication Protocol**

The communication protocol implemented by the TIELT – Call to Power 2 system may not have been entirely apparent from the list of messages used above. This section will examine the protocol from a broader perspective.

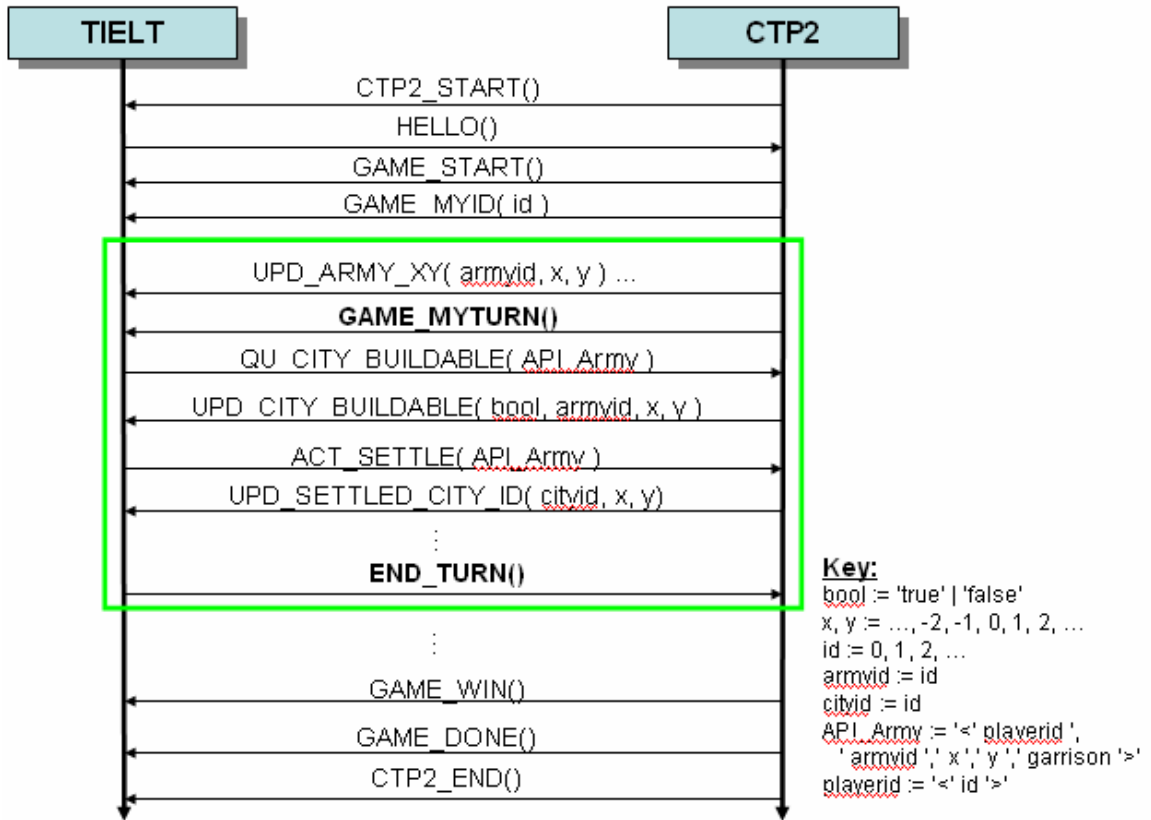
When Call to Power 2 is launched, it asks the user to begin the experiment in TIELT. When it detects a connection on the specified port, it sends a CTP2\_START( ) message to TIELT. TIELT responds by sending HELLO( ). Once an individual scenario has begun, Call to Power sends the GAME\_START( ) message. At this point, TIELT changes the internal state of the game to “Playing”. Then after the scenario is loaded, Call to Power 2 sends the GAME\_MY\_ID( ) message to inform TIELT of the ID of the player it is controlling. At this point the game may begin.

At the start of any of the player’s turns, Call to Power 2 will send UPD\_ARMY\_XY( ) messages to indicate the positions of all the player’s units. After this state update, the GAME\_MYTURN( ) message is sent to TIELT indicating that any desired actions can now be sent. As an example, to settle a city, TIELT would issue the

QU\_CITY\_BUILDDABLE message for one of its Settlers. If the Settler can build a city on its current location, Call to Power 2 will respond by setting the first argument of UPD\_CITY\_BUILDDABLE( ) to true and send this message to TIELT. TIELT sees that the action is possible and automatically responds with the ACT\_SETTLE( ) message for this Settler. After the city has been built, Call to Power 2 responds with the UPD\_SETTLED\_CITY\_ID( ) message to indicate the integer ID and location of the newly built city. Actions like this continue until the agent wishes to end the current turn, at which point it sends the END\_TURN( ) message.

Turns continue in this manner until a game is either won or lost. If, for example, the game is won, Call to Power 2 will send the GAME\_WIN( ) message to TIELT. Then after the scenario is closed, it will send the GAME\_DONE( ) message to TIELT, and TIELT will change the game state from “Playing” to “NotPlaying”. Finally, once the program is closed, the CTP2\_END( ) message is sent to TIELT to terminate the experiment. The example just described is modeled below in Figure 23. The green box in Figure 23 indicates all the actions taken in a single turn.





**Figure 23: TIELT – CTP2 Communication Protocol**

## 5. Sample Session

Now we can show a sample session of the game played with the fully integrated TIELT – Call to Power 2 system.

First, the system is connected and the experiment started. Note that the Connected() observation model was triggered, indicating that the game has started.

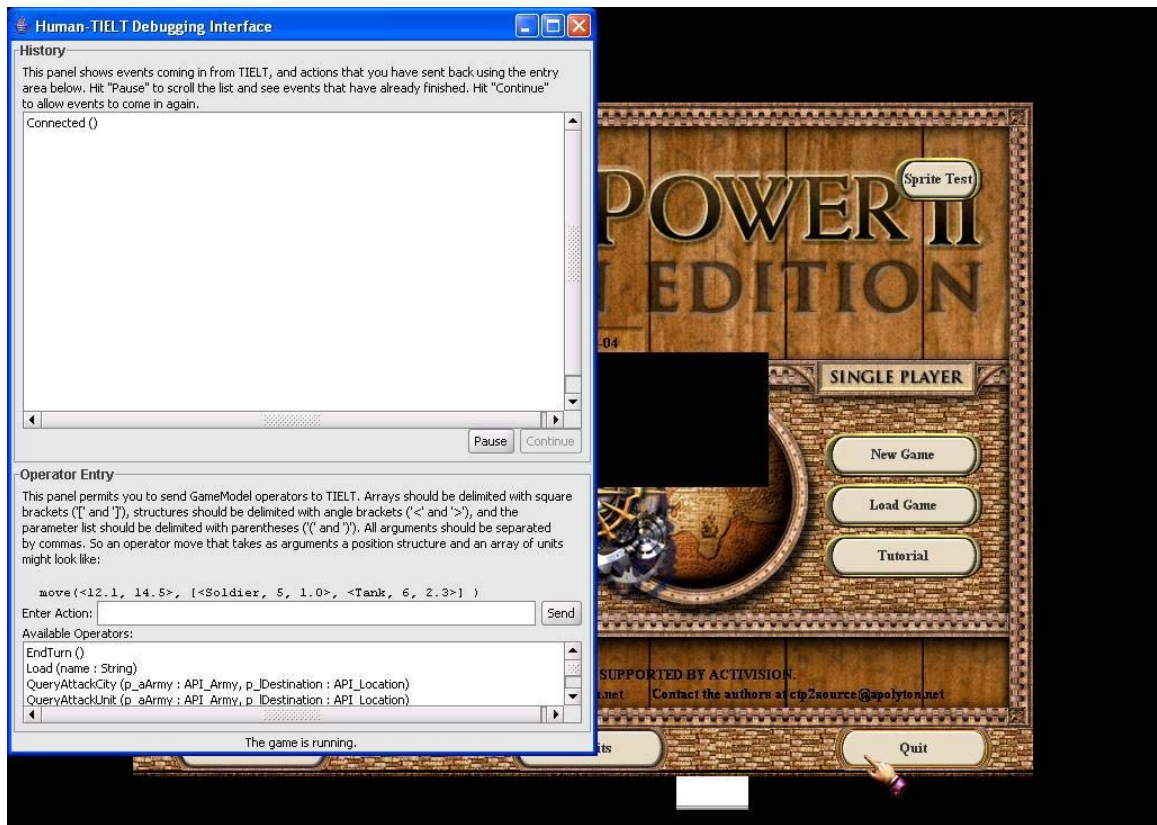
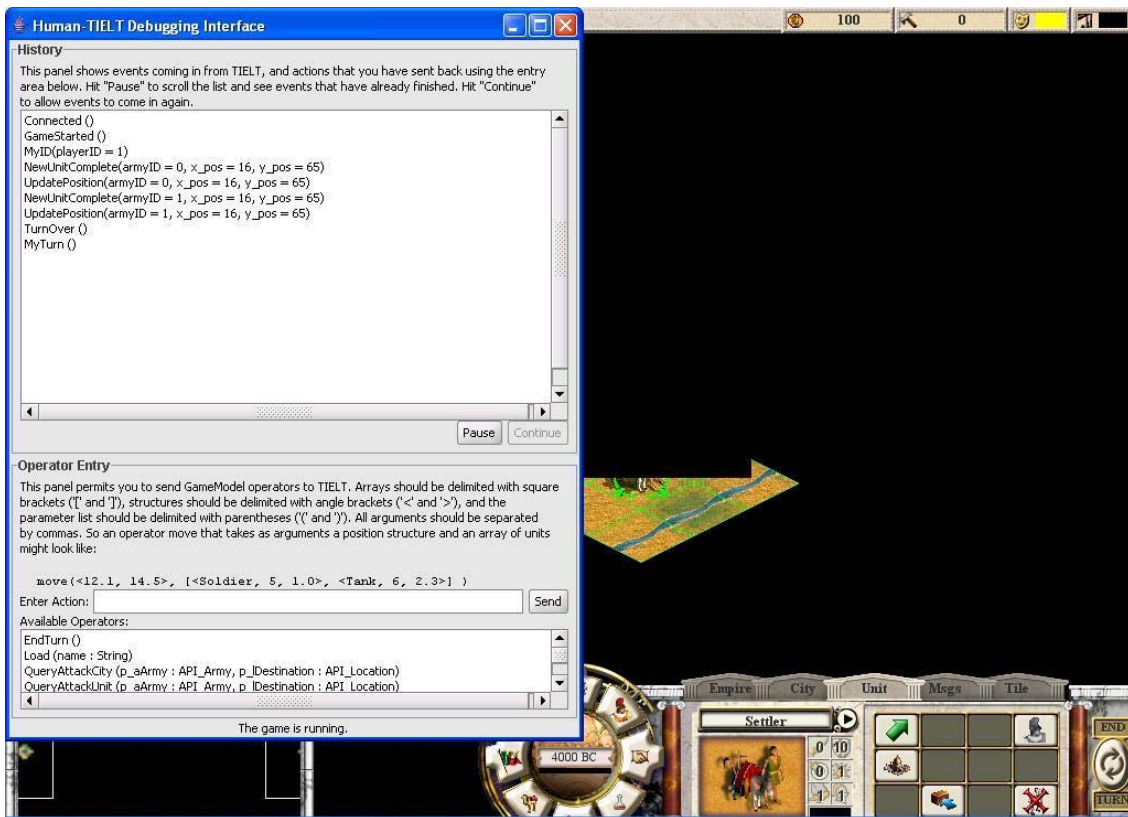
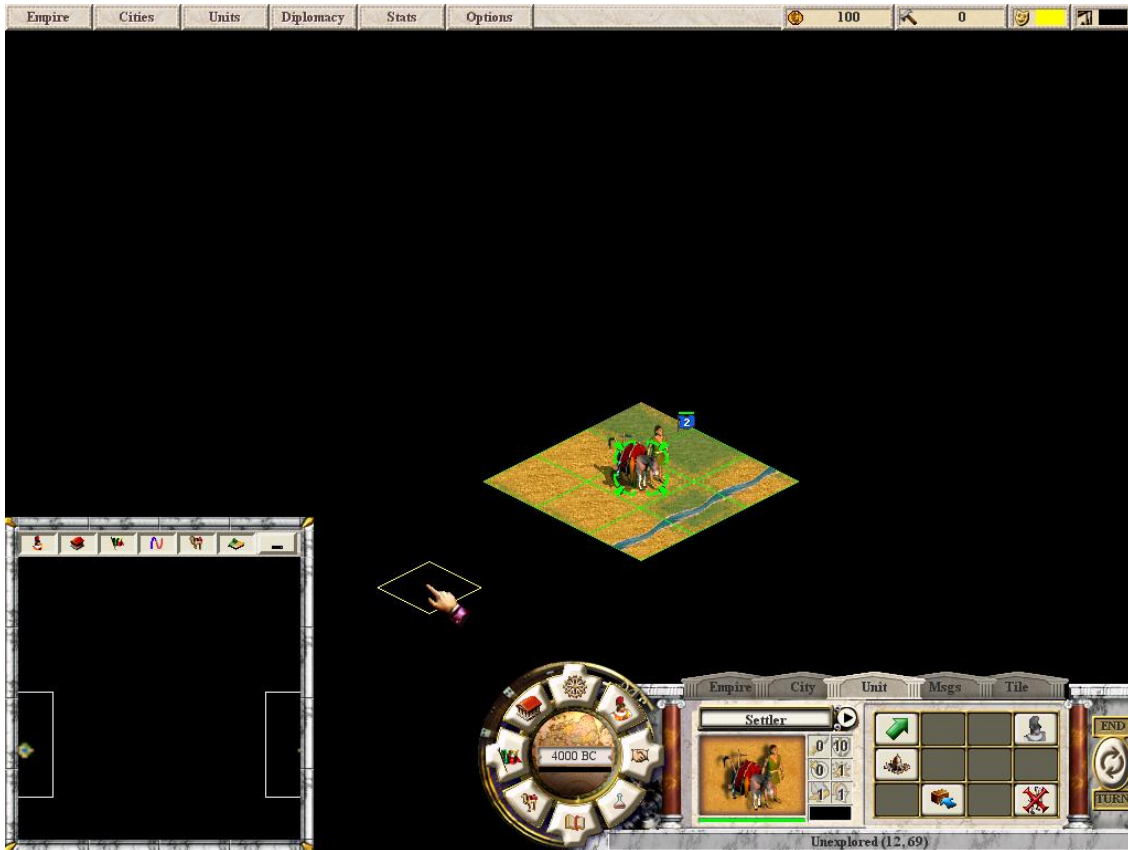


Figure 24: Start of Enhanced Session

Then an individual scenario is started and we receive state updates at the start of our turn indicating that we have two settler units with IDs of 0 and 1 at map location (16, 65).



**Figure 25: First Turn with Enhanced Session**

We attempt to move the settlers to the locations we desire, in this case (15, 65) and (18,65) with the QueryMoveable() action model. Then we settle new cities by sending the QueryCityBuildable() action model. Once these cities are settled, we receive the CitySettled() observation models to indicate the IDs and positions of these cities. Note that since settling a city destroys the Settler, we also receive a MyUnitDestroyed() message indicating the ID of the Settlers that were destroyed.

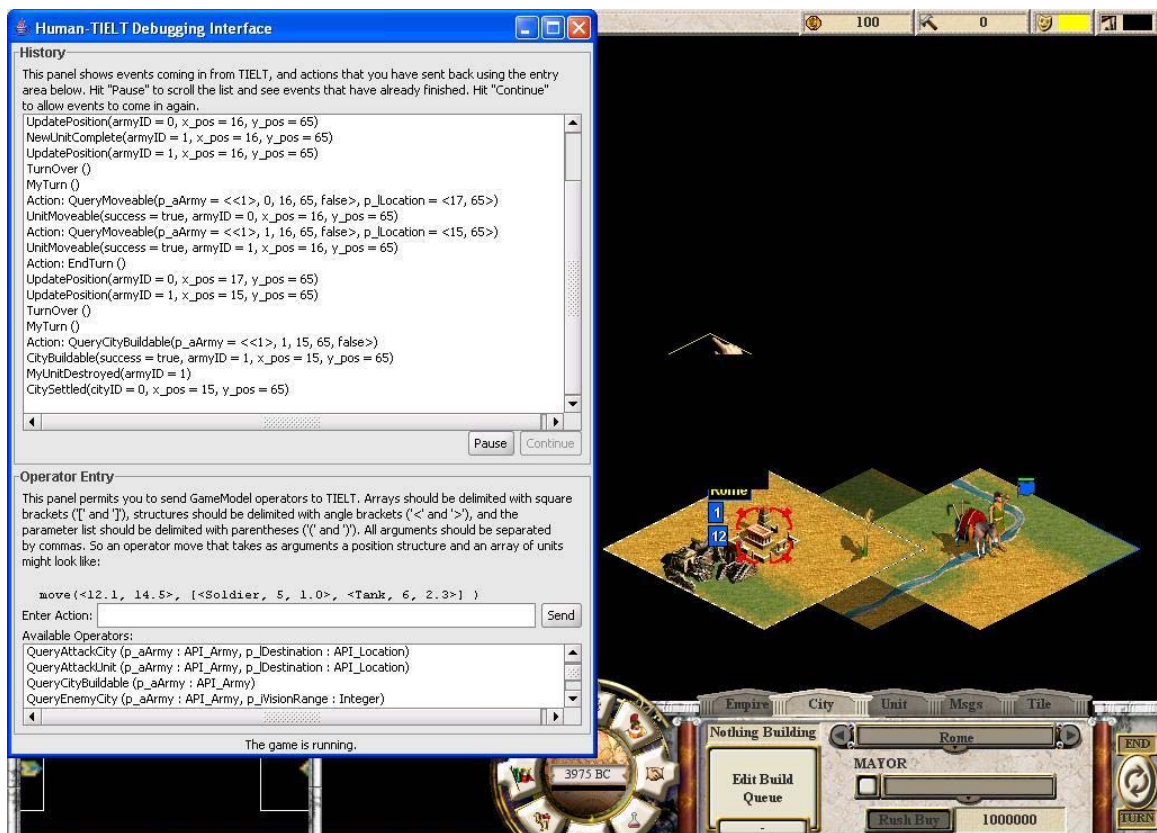
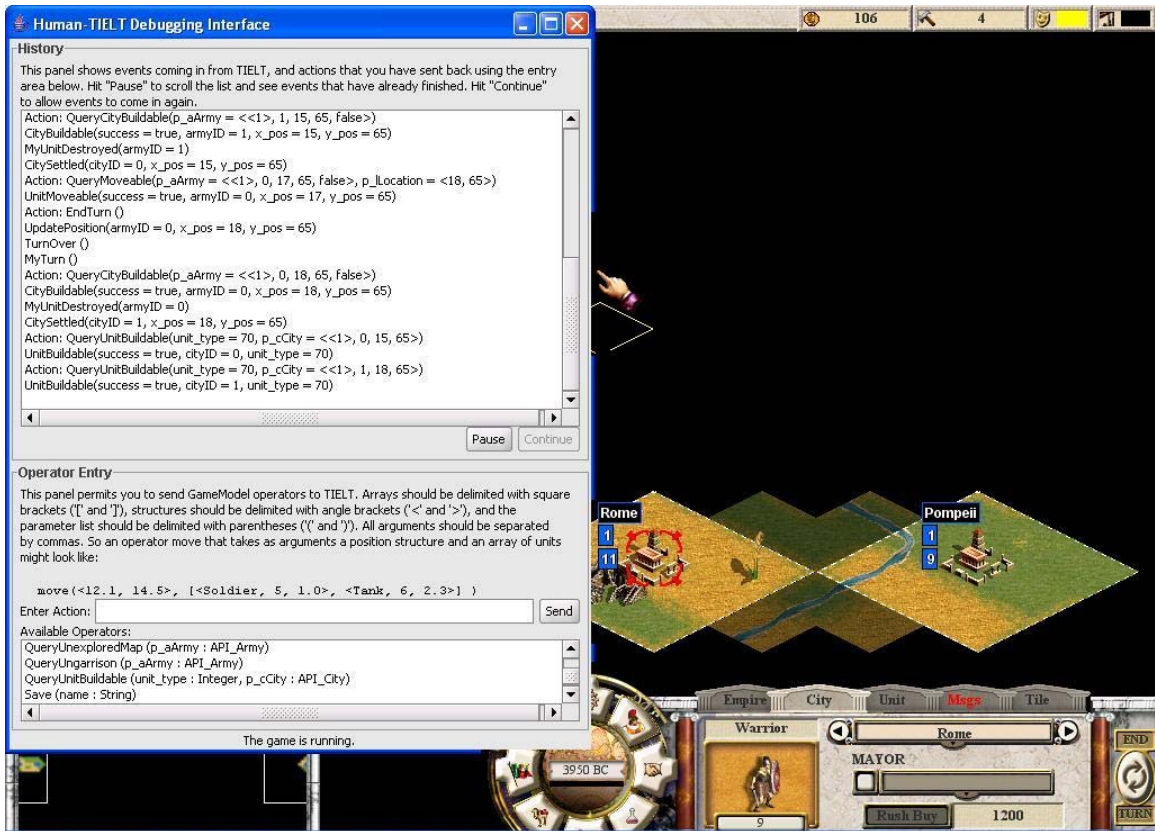


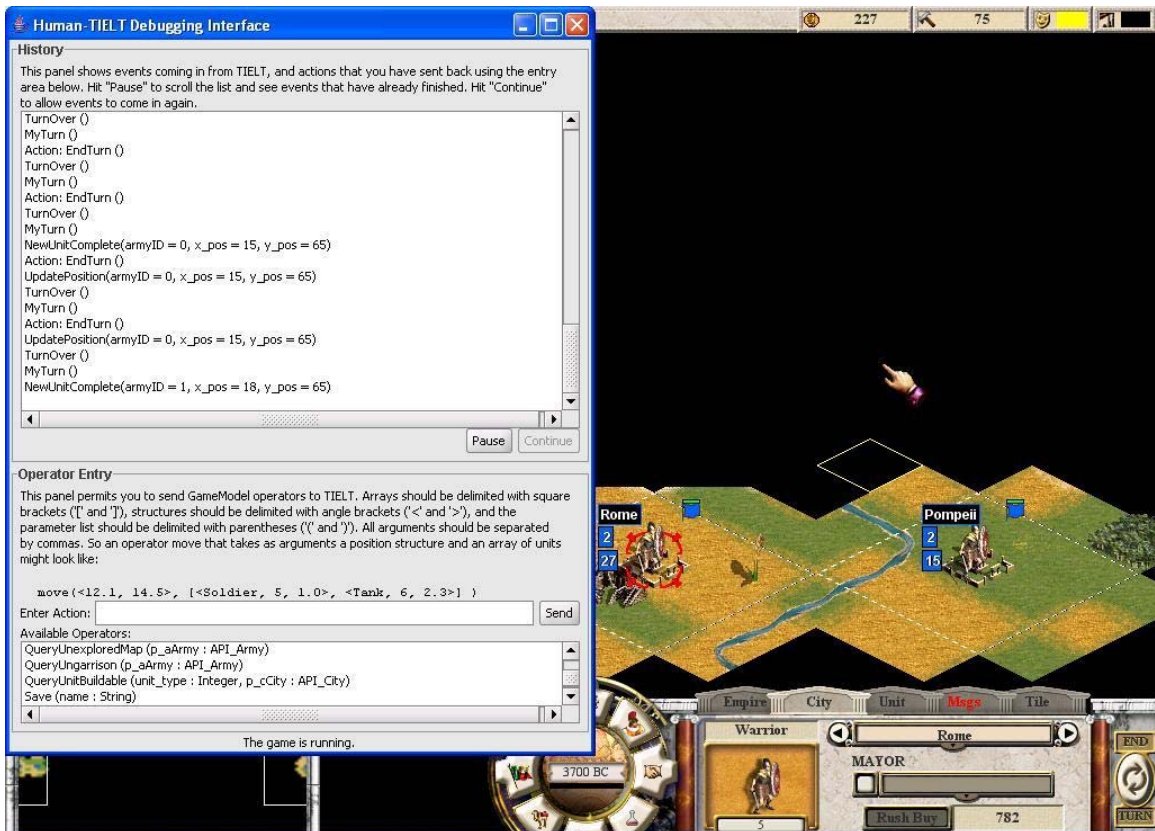
Figure 26: Building a City

We attempt to build Warrior units in these cities with the QueryUnitBuildable() action model. Note that 70 is the internal identification given to indicate Warrior units. We receive the UnitBuildable() updates with success set to true indicating that these units are now being built in the build queue.



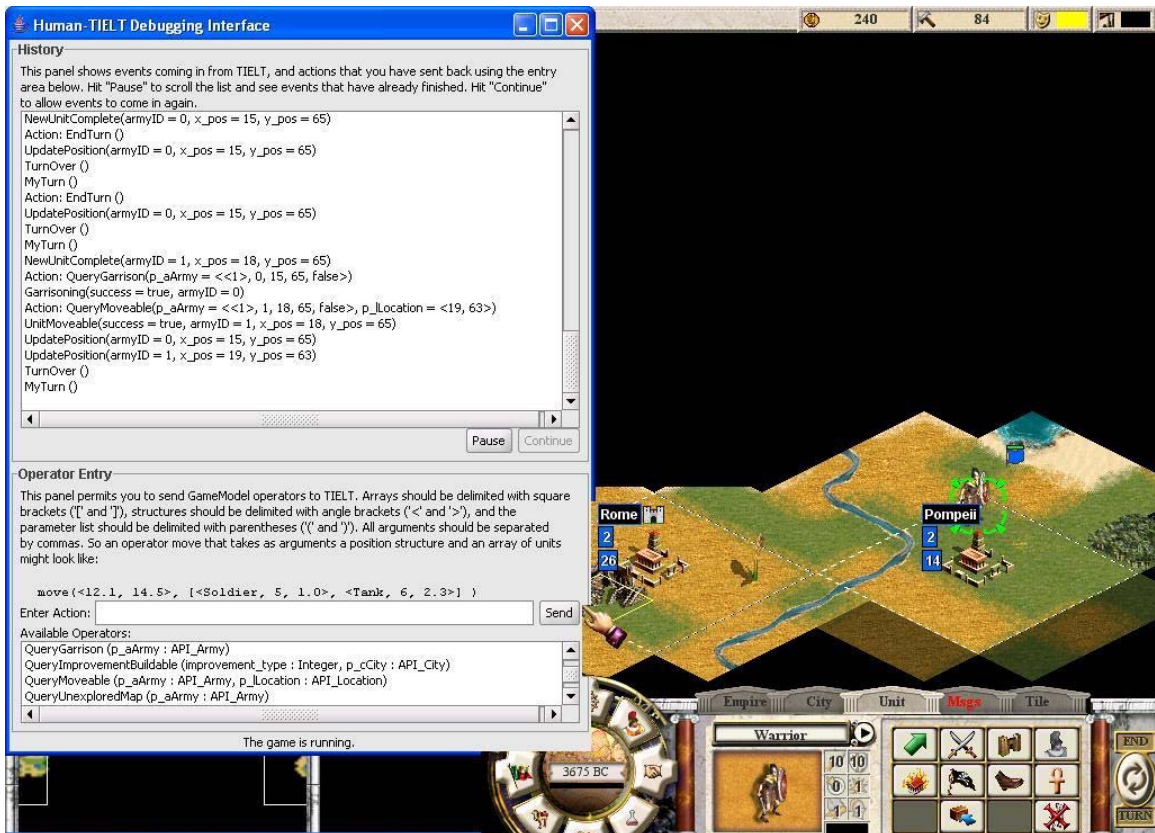
**Figure 27: Producing Military Units**

We let a few turns pass by sending the EndTurn() action model. Within a few turns the warriors are finished building. The NewUnitCompleted() observation model is triggered for each successfully built unit indicating the unit's ID and position.



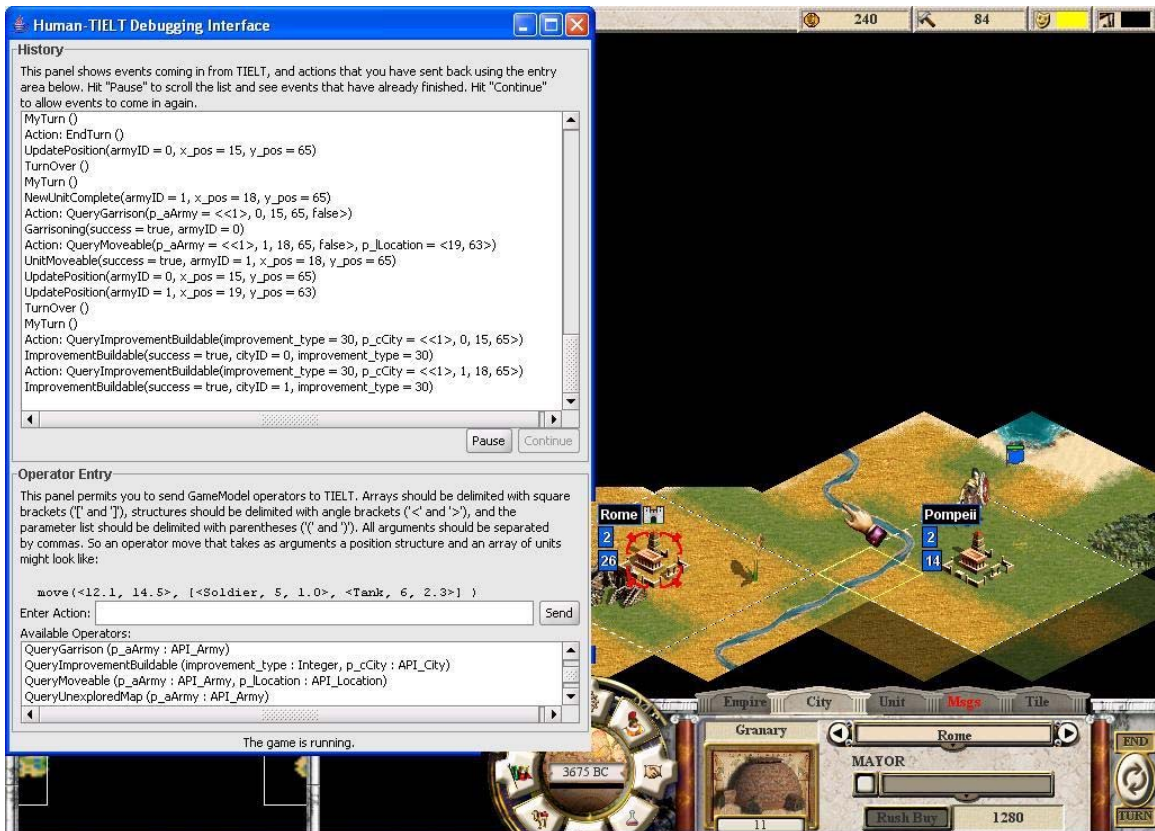
**Figure 28: New Units Completed**

We now attempt to have one warrior garrison its current location by issuing the QueryGarrison() action model, and have the other Warrior move to location (19,63) to explore the surrounding area.



**Figure 29: Exploring the Map**

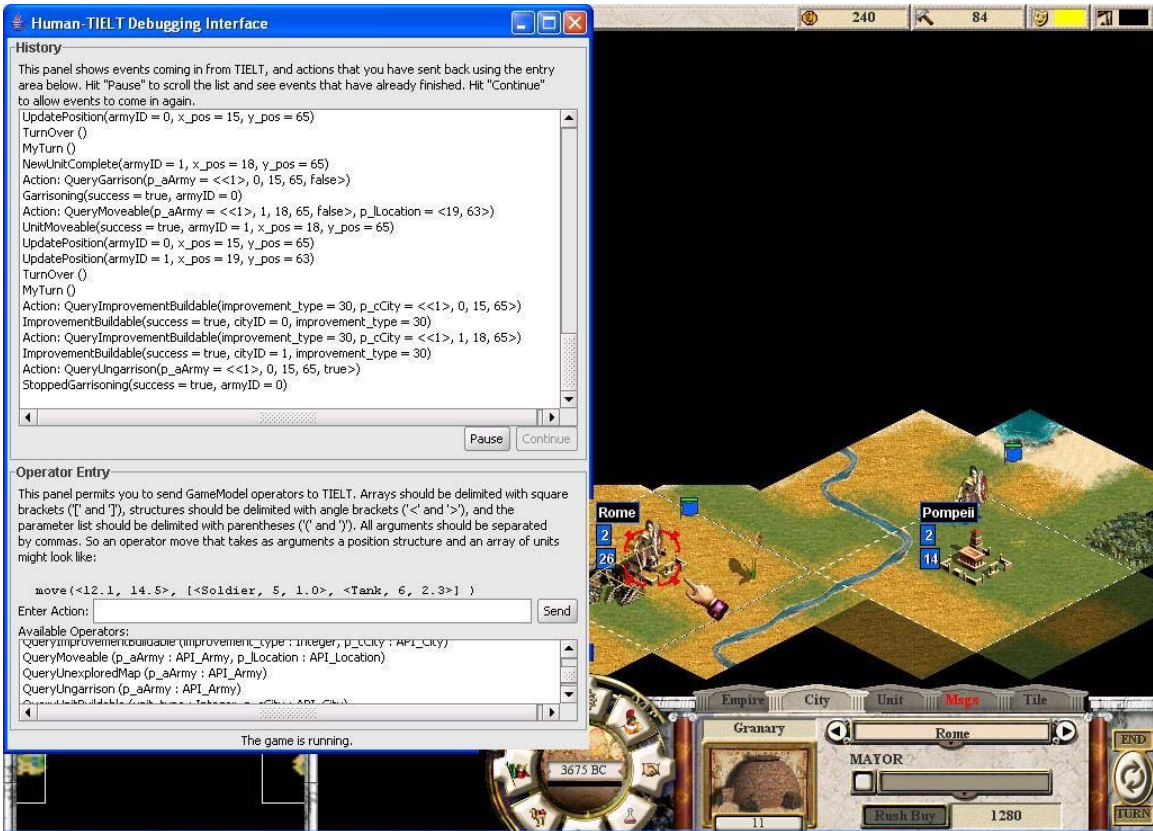
We now add granary (a city improvement) to the build queue in each city to help its production by issuing the QueryImprovementBuildable() action model. We receive the ImprovementBuildable() observation model with success set to true, indicating that the Granaries are in the build queue.



**Figure 30: Building City Improvements**

Next we will tell our garrisoning Warrior to ungarrison with the QueryUngarrison() action model so that we can use it to explore the map. We receive the StoppedGarrisoning() observation model with success set to true indicating that it has successfully stopped garrisoning.

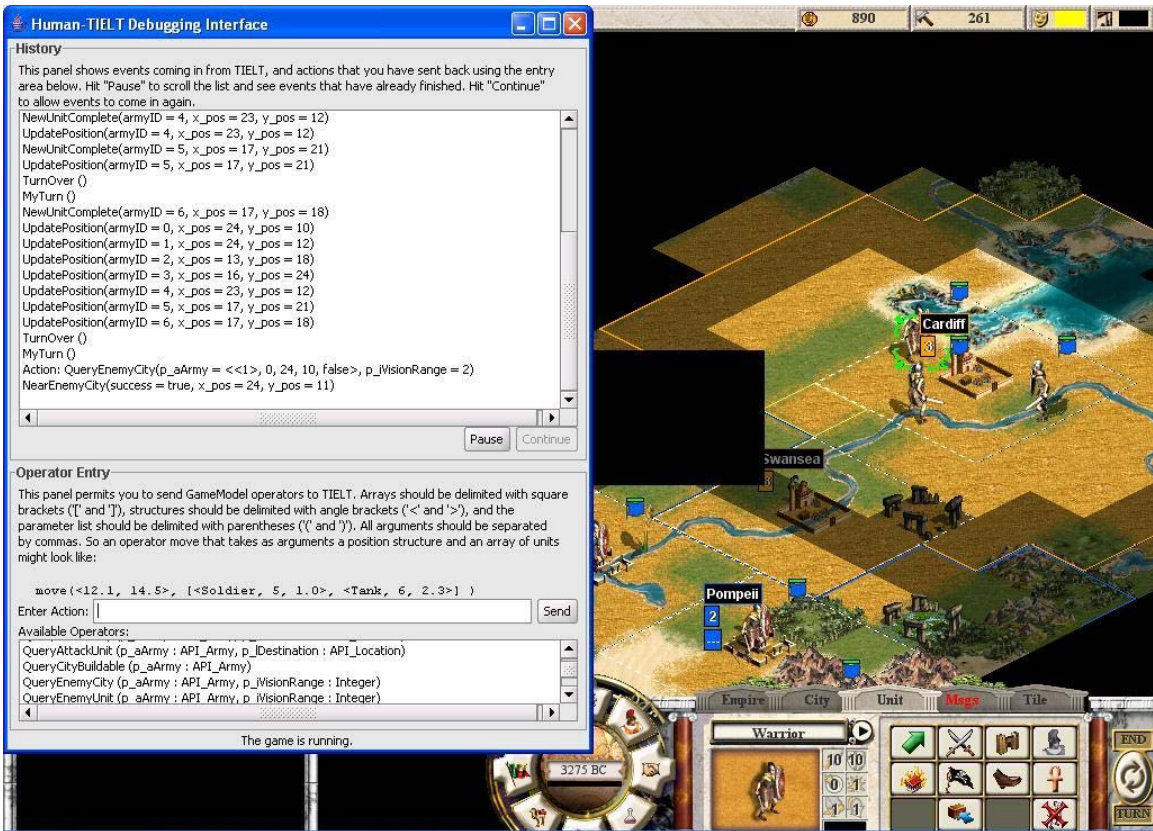




**Figure 31: Defending a City**

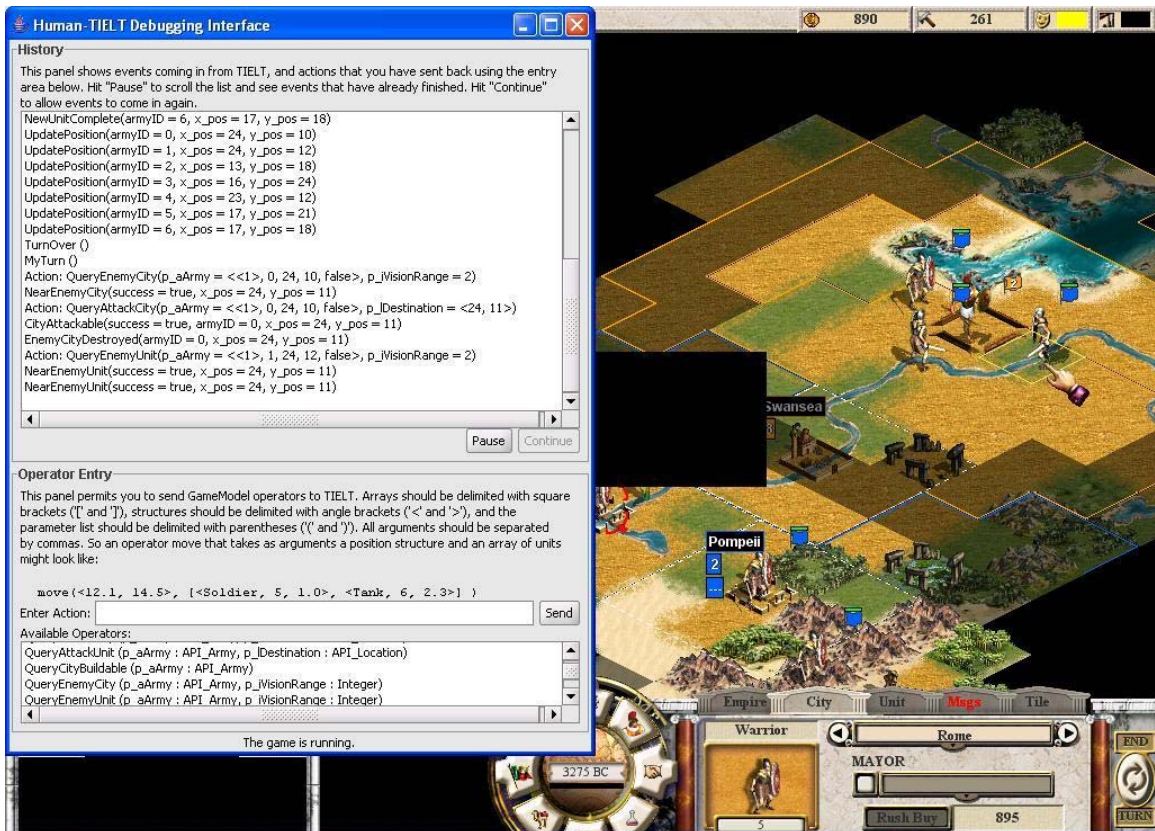
Advancing later into the game, we have more units and have explored more of the map.

We issue a query for enemy cities within visible range with the QueryEnemyCity() action model with our Warrior on (24, 10), and receive the NearEnemyCity() observation model telling us that there is an enemy city at (24, 11) on the map.



**Figure 32: Attacking an Enemy City**

Next we attempt to attack the enemy city that has been discovered at (24, 11) by issuing the `QueryAttackCity()` action model. The attack succeeds and we receive the `EnemyCityDestroyed()` observation model indicating that the city at (24, 11) has been destroyed. Then we query for remaining enemy units with the `QueryEnemyUnit()` action model. We receive two `NearEnemyUnit()` observation models for the enemy Hoplite and Settler on (24, 11).



**Figure 33: Attack Success**

We attempt to attack the enemy Hoplite with the QueryAttackUnit() action model. Our Warrior dies in the process and we receive the MyUnitDestroyed() observation model indicating that the Warrior with armyID 1 has been destroyed.

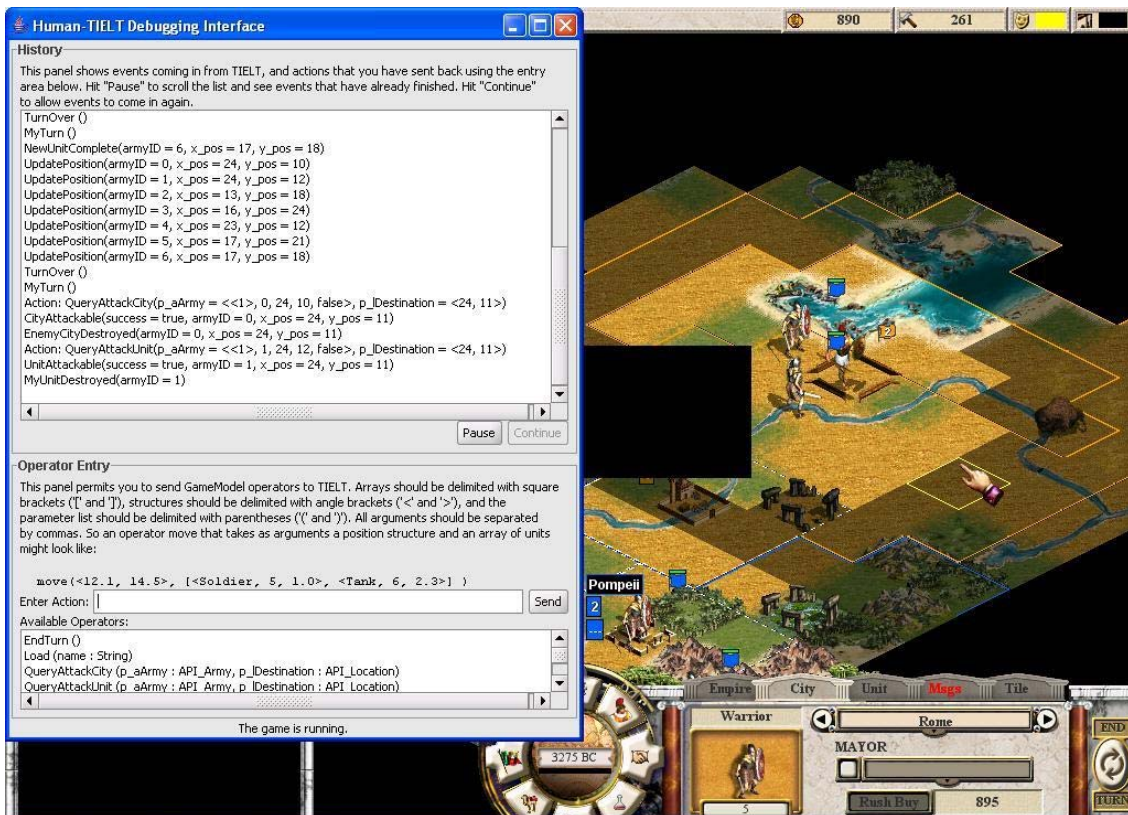


Figure 34: Attacking Enemy Unit

## **6. Conclusions**

### **6.1 Summary**

This thesis has detailed the work required to integrate Call to Power 2 with TIELT. A model of the rules of the game world was first defined within TIELT, and the desired actions and percepts an agent can take to play a complete subset of the game were specified. Then the existing Call to Power 2 API was modified and enhanced in order to both allow an outside agent to play a complete subset of the game and to integrate communications with TIELT. The work done for this project will provide an excellent test bed for future experiments in the Transfer Learning Project. With the completion of this thesis, intelligent agents can now be evaluated in conjunction with Call to Power 2. The next step for this project will be to design, implement, and test an assortment of transfer learning agents with the fully integrated TIELT – Call to Power 2 system.

### **6.2 Future work**

The Call to Power 2 – TIELT Integration has laid the groundwork for a multitude of later projects. As stated, we now have a test bed for evaluating the performance of intelligent agents in conjunction with the DARPA Transfer Learning Project. Our next milestone with this project is to construct agents that perform reasoning upon the current state of the world, as modeled in TIELT, in order to evaluate an assortment of transfer learning algorithms. Additionally, as our API and TIELT specifications are all freely available, any AI researcher or hobbyist can make use of any part of this project for their own endeavors.

## Bibliographical References

Aha, D. (2007) *Testbed for Integrating and Evaluating Learning Techniques*.

URL: [http://www.tielt.org/presentations/TIELT Project Overview \(17 Nov 2004\).ppt](http://www.tielt.org/presentations/TIELT Project Overview (17 Nov 2004).ppt).

Last checked: 4/16/07

Apolyton (2007).

URL: <http://apolyton.net/forums/forumdisplay.php?s=&forumid=213>]. Last checked:

4/16/07

E. L. Thorndike & R. S. Woodworth (1901). The Influence OF Improvement in one Mental Function upon The Efficiency of other functions (I). *Psychological Review*, 8, 247-261. URL: <http://psychclassics.yorku.ca/Thorndike/Transfer/transfer1.htm>. Last checked: 4/16/07

Gudevia, U. (2006). *Integrating War Game Simulations with AI Testbeds: Integrating Call To Power 2 with Tielt*. Computer Science and Engineering. Lehigh University.

Transfer Learning (2006). URL: <http://www.darpa.mil/baa/BAA05-29.html>. Last checked: 4/16/07.

## **Vita**

Joseph Henry Souto was born in Morristown, NJ to Joseph and Pamela Souto. He attended Lehigh University from the fall of 2000 through the spring of 2004, receiving a B.S. in Computer Engineering with honors. He returned to graduate school at Lehigh University in the fall of 2005 and will receive his Master's in the spring of 2007.