Scientific Community Game (SCG) Court User Guide

Anush Prabhu Ramachandran
Daniel Marcucci
Rashmi Nayak

Table of Contents

1 Introduction

For the objective of the CS 5500 / Managing Software Development course, we want to develop community support for running competitions of games that involve computational problems. In order to achieve this goal, a consistent interface was developed so that playground designers may create unique games in an effective and efficient way. This guide describes the use of this system by specifying [1] the general procedures for scientific community games, [2] the processes for defining domain, protocols, and playgrounds, [3] signing up, enrolling, and registering avatars capable of playing the games detailed within playground designs, [4] managing and running game tournaments amongst avatars, and [5] analyzing the histories of previously completed tournaments. Additionally, flaws and issues of the current implementation are discussed and potential solutions for future refinements are suggested.

2 Generic SCG Procedures

Regardless of how a playground is configured, the generic scientific community game procedures remain consistent. Specifically, the performance evaluations of competing avatars always result in a computation of updates to their reputations. These performance evaluations take place after the completion of an opposition process (i.e., refutation, strengthening, or agreement). The following sections describe the different types of oppositions in detail and explain how reputation is handled at an abstract level.

2.1 Use of Reputation

When a tournament begins, the reputation for each of the avatars is equal. This initial reputation value can be configured by the playground designer using the configuration file constant.

Throughout the duration of a tournament, a number of reputation updates will take place. In each update, reciprocal changes are made to the reputations of two avatars (i.e., a winner and a loser of a particular opposition). Therefore, reputation is zero-sum (because the winner gains the same amount that is deducted from the loser).

The following section describes the steps that lead to determining what reputation updates need to be made and which avatars those updates should be applied to.

2.2 Opposition Process

(NOTE: For the purposes of this documentation, always assume that '*Alice*' is the person making the claim and that '*Bob*' is the person acting on / opposing the claim.)

During a tournament, avatars must both (A) propose new claims and (B) act on (i.e., oppose) the claims that have been proposed by other avatars. When an avatar opposes a claim, it must choose whether to [1] refute it, [2] strengthen it, or [3] agree with it. For each of these actions, there are a number of steps that must be carried out by the proposer (i.e., '*Alice*') and the opposer (i.e., '*Bob*'). These steps vary based on the type of opposition and the protocol of the claim. The tournament admin (a component of the SCG court system software) ensures that all the steps of the protocol are carried out properly while the outcome of the protocol is determined based on the protocol predicate.

2.2.1 Refutation

Alice makes a claim C using protocol P with quality q^A and confidence cf^A. Bob refutes C.

The protocol P specifies the sequence of actions that are to be to be performed by Alice and Bob.

Depending on the protocol, either Alice or Bob (or both) will provide an instance to be solved and a solution for the given instances of claim C. The quality of the solution(s) and quality q^A will be used by the protocol predicate (i.e., *getResult*) to determine the outcome of the refutation (see Section 4 for more details). The result of this predicate is a value between 1 and -1. This result is used in computing the updates to the reputation of the two avatars.

If Bob successfully refutes the claim, Bob wins reputation and Alice loses reputation.
If Alice successfully defends her claim, Alice wins reputation and Bob loses reputation.

The reputation is updated as below:

Alice's new reputation = Alice's current reputation + (cf^A * result)

Bob's new reputation = Bob's current reputation - (cf^A * result)

NOTE**:**

result: Output of the Protocol predicate – getResult . This is a value between -1 and +1. 1 indicates refutation fails (i.e., Alice wins). -1 indicates refutation succeeds (i.e., Bob wins). Any value between -1 and 1 indicates that the partial success of the refutation

2.2.2 Strengthening

Alice makes a claim C using protocol P with quality q^A and confidence cf^A.

Bob strengthens claim C with quality q^B and cf^B where $cf^B >= cf^A$. Alice refutes this strengthened claim. The refutation follows the steps specified in the protocol P.

If Bob successfully defends his strengthened claim, Bob wins and the reputation updates are as follows:

Bob's new reputation = Bob's current reputation + ($cf^A * | q^A – q^B |$)

Alice's new reputation = Alice' current reputation - ($cf^A * | q^A - q^B |$)

If Bob fails to defend his strengthened claim, Alice wins and the reputation updates are as follows:

Alice's new reputation = Alice's current reputation + cf^B

Bob's new reputation = Bob's current reputation – cf^B

2.2.3 Agreement

Alice makes a claim C using protocol P with quality q^A, confidence cf^A.

When Bob agrees on claim C with Alice, the following conditions should hold true:

Bob must defend C against Alice.

Bob must refute !C (i.e., the negated claim of C). !C has the same InstanceSet, quality and confidence as C but has a protocol !P with Alice as defender.

If Bob fails to satisfy any one of the above condition, then Bob loses.

Similarly Alice must satisfy the following conditions:

Alice must defend C against Bob.

Alice must refute !C with Bob as the defender.

If Alice fails to satisfy any one of the above condition, then Alice loses.

If Bob loses, the reputation is updated as follows:

Bob's new reputation = Bob's current reputation – cf^A

Alice's new reputation = Alice's current reputation + cf^A

If Alice loses, the reputation is updated as follows:

Alice's new reputation = Alice's current reputation – cf^A

Bob's new reputation = Bob's current reputation + cf^A

If both Alice and Bob satisfy all their conditions, the reputations remain unaffected and the claim goes into the social welfare set (i.e., the claim repository).

3 Domain Definitions

Designers that want to create games using computational problems can employ the scientific community game (i.e., "SCG") court system by authoring domain definitions. A domain definition consists of class definitions that represent how instances, claims, and solutions of a computational problem are constructed. Additionally, a domain definition includes the specification of rules for [1] deciding if a solution is valid for a particular instance, [2] calculating the quality of a valid solution, and [3] determining if a claim belongs to an instance. Finally, the designers must provide functionality that will serve as the foundation for creating basic avatars (i.e., "baby avatars") that are unsophisticated but capable of competing in tournaments without violating the game rules.

(NOTE: In the subsequent sections, it is suggested that calls to classes, functions, and file names starting with "DDS" are changed to start with a three letter representation of the domain that is being designed. While this suggestion is intended to simplify the process from converting the provided templates into a domain specific definition, it is not necessary to ensure functionality with the SCG court system.)

3.1 Domain Class Definition

As mentioned, a particular domain design is comprised of class definitions that specify the necessary elements of a corresponding computational problem. These class definitions act as representations of the computational problem as a scientific community game in which [1] instances are proposed, [2] claims are made, and [3] solutions are provided, validated and computed. The following sections describe how to create domain class definitions by explaining the various required components and providing code samples (both at a generic and specific level).

3.1.1 Overview

When a designer wants to formalize a domain, they will need to define four **d**omain **d**esign **s**pecific (i.e., "`dds`") classes: [1] instance, [2] solution, [3] instance set, and [4] configuration:

`ddsInstance`: Represents a single instance of a problem (i.e., a concrete question which can be solved by the game players).

`ddsSolution`: Represents a solution corresponding to a given problem instance.

`ddsInstanceSet`: Represents a set/collection of problem instances in our domain for which claims can be made (i.e., the abstract question to be solved).

`ddsConfig`: Represents the customizable constraints for this domain, which is part of a playground configuration (see Section 5.2).

These definitions are contained in a domain design specific class dictionary (i.e., "ddsDomain.cd").

3.1.2 Code Sample

The code sample below is a template class dictionary (i.e., "ddsDomain.cd") that domain designers can modify to specify their scientific community games. Within this code, comments (provided in green) explain the purpose of each line and/or what modifications are required when a domain designer is attempting to make changes. Additionally, it is interesting to note that this template, when coupled with the corresponding behavior file (.beh) template, will serve as a complete definition of the domain.

References to domain design specific data are denoted in orange (i.e., dds). During actual implementation, this text should be replaced with a three letter representation that distinguishes a particular domain. For example, an Instance in the Highest Safe Rung (i.e., hsr) domain would be called an hsrInstance. Therefore, a domain designer for the Highest Safe Rung problem would replace all dds text in the template below with hsr. Note that the class dictionary file name should also reflect this change. Therefore, the resulting file in this example would be named "hsrDomain.cd" if it were being created for the Highest Safe Rung problem.

Finally, definitions containing "[...]" should be replaced with unique and valid class definition components (see Section 3.1.3 for a specific example). For general information on class dictionary and behavior files, please refer to the DemeterF documentation.

```
/*
 * File: ddsDomain.cd
 * (replace all instances of dds, including the file name, with
 * a three letter representation of the domain name)
 */

/*
 * "scg.cd" includes the class definitions for SCG courts.
 *
 * The SCG class dictionary has meta-level definitions
 * that specify how each domain level cd must be defined.
 */

nogen include "../scg/scg.cd";

/*
 * This domain definition will be created within the dds
 * package.
 */

package dds;

/*
 * Import the SCG level classes and interfaces.
 *
 * These are necessary to ensure that the domain level cd is
 * compatible with the SCG courts system.
 *
 * (After importing the SCG package, you should also import any
 * additional packages or classes that will be required for the
 * implementation of methods in your .beh file)
 */

import scg.*;
// Enter additional 'import' statements before this comment.

/*
 * NOTE: In the following definitions, you must replace "[...]"
 * with your unique & valid class definition components (see
 * Section 3.1.3 below for specific examples)
 */

/*
 * Define a domain design specific Instance.
 *
 * The ddsInstance definition must implement the InstanceI
 * interface. Therefore, InstanceI methods must be defined in
 * your corresponding behavior file (.beh) as part of the
 * ddsInstance class.
 */

ddsInstance = "[...]" implements InstanceI.

/*
 * Define a domain design specific Solution.
 *
 * The ddsSolution definition must implement the SolutionI
 * interface. Therefore, SolutionI methods must be defined in
 * your corresponding behavior file (.beh) as part of the
 * ddsSolution class.
 */

ddsSolution = "[...]" implements SolutionI.

/*
```

```
 * Define a domain design specific InstanceSet.
 *
 * The ddsInstanceSet definition must implement the InstanceSetI
 * interface. Therefore, InstanceSetI methods must be defined in
 * your corresponding behavior file (.beh) as part of the
 * ddsInstanceSet class.
 */

ddsInstanceSet = “[...]” implements InstanceSetI.

/*
 * Define a domain design specific Config.
 *
 * The ddsConfig definition must implement the DomainConfigI interface.
 *
 * Each ddsConfigurableVariableX (where X is a positive number) should
 * be replaced with a unique variable name (see Section 3.1.3 for a
 * specific example).
 *
 * (NOTE: For a specific domain design, you may add an arbitrary amount
 * of dds configurable variables. For the purposes of this
 * documentation, however, only 2 are used.)
 */

ddsConfig = “dds_config["
                          *l*t “ddsConfigurableVariable1:” *s “[...]”
                          *l*t “ddsConfigurableVariable2:” *s “[...]”
                          *l
                           “]” implements DomainConfigI.
```

3.1.3 Example

The example provided below describes a domain design for the **c**onstraint **s**atisfaction **p**roblem (i.e., “CSP”). Therefore, these class definitions are included in a file named “cspDomain.cd“. Comments (provided in green) explain the changes (provided in light blue). While these comments are unnecessary when actually creating a domain design, they are provided in this document to highlight what modifications were made from the original template (in Section 3.1.2).

```
// File: cspDomain.cd

// This line is the same in all domain designs:
```

```
nogen include "../scg/scg.cd";

// Changed package name from 'dds' to 'csp':
package csp;

// This line is the same in all domain designs:
import scg.*;
// Added import of the Iterator class for use in cspDomain.beh:
import java.util.Iterator;

/*
 * - Changed from 'ddsInstance' to 'cspInstance'
 * - Replaced "[...]" with CSP instance specific classes
 * - Maintained 'implements InstanceI' declaration
 */
cspInstance = <vars> List(Var) *s <clauses> Cons(Clause) implements
InstanceI.

Clause = "(" <relnum> int *s "{" <weight> int "}" *s <vars> List(Var)
")".

/*
 * - Changed from 'ddsSolution' to 'cspSolution'
 * - Replaced "[...]" with CSP solution specific classes
 * - Maintained 'implements SolutionI' declaration
 */
cspSolution = <assign> ListMap(Var,Boolean) implements SolutionI.

Var = <id> ident implements Comparable(Var).

/*
 * - Changed from 'ddsInstanceSet' to 'cspInstanceSet'
 * - Replaced "[...]" with CSP instance set specific classes
 * - Maintained 'implements InstanceSetI' declaration
 */
cspInstanceSet = "(" <type> ListSet(Integer) ")" implements
InstanceSetI.

/*
 * - Changed from 'ddsConfig' to 'cspConfig'
 * - Replaced "[...]" with CSP specific config parameters
 * - Maintained 'implements DomainConfigI' declaration
 */
cspConfig = "csp_config["
            //maximum value of relation number in Instance allowed
```

```
                        *l*t "maxRelNum:" *s <maxRelNum> int
                        //maximum number of variables allowed in Instance
                        *l*t "maxVariables:" *s <maxVariables> int
                         *l
                         "]" implements DomainConfigI.
```

3.2 Domain Behavior Definitions

For every domain design specific class dictionary, there must be an equivalent domain design specific behavior definition that specifies the particular rules of the domain and how to compute solution qualities. This functionality is used by the tournament admin to validate the legitimacy of game actions and determine the quality of the solution. Additionally, a domain behavior definition illustrates how to decide whether or not an instance corresponds to a provided instance set. The following sections describe how to create domain behavior definitions by explaining the various required components and providing code samples (both at a generic and specific level).

3.2.1 Overview

When a designer wants to formalize a domain, they will need to define certain methods from SCG interfaces:

`double` **`valid`**`(SolutionI solution)`: The “valid” method is used to check if the solution provided by the player is valid with respect to this Instance object. Defined in the “`ddsInstance`” class, it returns a double and is given a SolutionI. It returns 1 if the solution is valid or 0 if the solution is invalid.

`double` **`quality`**`(SolutionI solution)`: The “quality” method is used to calculate the quality of the solution provided for this Instance object. Defined in the “`ddsInstance`” class, it returns a double and is given a SolutionI. It returns the quality as double between 0 to 1 (with 0 being the least quality and 1 being the max quality).

`Option<String>` **`belongsTo`**`(InstanceI instance)`: The “belongsTo” method checks if the instance provided by the player corresponds to this InstanceSet. Defined in the “`ddsInstanceSet`” class, it returns an Option<String> with an error string if the instance does not belong to this InstanceSet or a None<String> if there is no error.

Additionally, certain functionality should be defined as part of the `ddsConfig` class (see Section 3.2.2). Parameters of the `ddsConfig` class should also be taken into account when designing the valid method.

These definitions are contained in a domain design specific domain behavior file (i.e., "`ddsDomain.beh`").

3.2.2 Code Sample

The code sample below is a template behavior definition (i.e., "`ddsDomain.beh`") that domain designers can modify to specify functions for their scientific community games. Within this code, comments (provided in green) explain the purpose of each line and/or what modifications are required when a domain designer is attempting to make changes.

References to domain design specific data are denoted in orange (i.e., `dds`). During actual implementation, this text should be replaced with a three letter representation that distinguishes a particular domain. For example, the Instance class in the Highest Safe Rung (i.e., `hsr`) domain would be the `hsrInstance` class. Therefore, a domain designer for the Highest Safe Rung domain would replace all `dds` text in the template below with `hsr`. Note that the behavior file name should also reflect this change. Therefore, the resulting file in this example would be named "`hsrDomain.beh`" if it were being created for the Highest Safe Rung problem.

Finally, any cases of `return null;` should be replaced with unique and valid Java code (see Section 3.2.3 a specific example).

```
/*
 * File: ddsDomain.beh
 * (replace all instances of dds, including the file name, with
 * a three letter representation of the domain name)
 */

/*
 * Methods for the ddsInstance class.
 *
 * The ddsInstance class must implement the InstanceI interface.
 * Therefore, the 'valid' and 'quality' methods must be
 * implemented.
 */
ddsInstance {{
// Checks if a solution is valid with respect to this ddsInstance
public double valid(SolutionI solution)
```

```
        {
               // Replace with domain specific logic:
               return null; // Must return a ‘double’ object
        }

        /*
         * Calculate the quality of a solution with respect to this
         * ddsInstance
         */
        public double quality(SolutionI solution)
        {
               // Replace with domain specific logic:
               return null; // Must return a ‘double’ object
        }

        /*
         * Include helper methods for the ‘valid’ and/or ‘quality’
         * methods here (i.e., before the double curly brackets)
         */

        }}

        /*
         * Methods for the ddsInstanceSet class.
         *
         * The ddsInstanceSet class must implement the InstanceSetI
         * interface. Therefore, the ‘belongsTo’ method must be
         * implemented.
         */
        ddsInstanceSet {{
        // Check if an instance belongs to this ddsInstanceSet
        public Option<String> belongsTo(InstanceI instance)
        {
               // Replace with domain specific logic:
               return null; // Must return an ‘Option<String>’ object
        }


        /*
         * Include helper methods for the ‘belongsTo’ method here
         * (i.e., before the double curly brackets)
         */

        }}

        /*
         * Customizable ddsConfig parameters for the dds domain. These must
```

```
 * match the information defined in the ddsConfig class (within the
 * ddsDomain class dictionary). Calls to [defaultValue] must be
 * replaced with an appropriate value.
 *
 * The ddsConfig class must define DEFAULT_DDS_CONFIG, which specifies
 * the default configuration values for the ddsConfig parameters
 * defined in the ddsDomain class dictionary.
 *
 * The methods getDefaultDomainConfig() and getDefaultConfig() must
 * also be implemented, which return DEFAULT_DDS_CONFIG and a default
 * Config respectively.
 *
 * (NOTE: For a specific domain design, you may add an arbitrary amount
 * of dds configurable variables (so long as it is consistent with the
 * ddsDomain class dictionary). For the purposes of this documentation,
 * however, only 2 are used.)
 */
ddsConfig {{
	private static ddsConfig DEFAULT_DDS_CONFIG;
	static {
		try {
			DEFAULT_dds_CONFIG = ddsConfig.parse(

			"dds_config[\n" +
			"ddsConfigurableVariable1: [defaultValue]\n" +
			"ddsConfigurableVariable2: [defaultValue]\n" +
			"]"
			);
		}
		catch(Exception ex)
		{
			ex.printStackTrace();
		}
	}

	public static ddsConfig getDefaultDomainConfig()
	{
		return ddsConfig.DEFAULT_DDS_CONFIG;
	}

	public static Config getDefaultConfig()
	{
		return new Config(SCGConfig.getDefaultSCGConfig(),
		getDefaultDomainConfig());
	}

}}
```

3.2.3 Example

The example provided below describes the rules and regulations for the **c**onstraint **s**atisfaction **p**roblem (i.e., "CSP"). Therefore, these behavior definitions are included in a file named "cspDomain.beh". Comments (provided in green) explain the changes (provided in light blue). While these comments are unnecessary when actually creating a behavior file, they are provided in this document to highlight what modifications were made from the original template (in Section 3.2.2). Additional comments (in black) explain the implemented code functionality.

```
// File: cspDomain.beh

/*
 * Methods for the cspInstance class.
 *
 * The cspInstance class must implement the InstanceI interface.
 * Therefore, the 'valid' and 'quality' methods must be
 * implemented.
 */
cspInstance {{
// Checks if a solution is valid with respect to this cspInstance
public double valid(SolutionI solution)
{
      // Cast the SolutionI object to a CSPSolution object
      CSPSolution cspSolution = (CSPSolution) solution;

      // If the number of assignments does not equal the number of variables...
      if ( cspSolution.getAssign().size() != getVars.length() )
            return 0; // ...return 0 (i.e., not valid)

      // Validation variable
      boolean isValid = true;
      // Iterator (to check if each var instance is contained in the assignment)
      Iterator<Var> vars = getVars().iterator();

      // For each var in the Iterator...
      for(vars; vars.hasNext();)
      {
       /*
        * check if the current var is accounted for in the assignment
        * and store the result of this check (and similar subsequent
        * checks) in the validation variable
        */
            isValid &= cspSolution.getAssign().containsKey(vars.next());
        }

        // Return the result of the findings
        if (isValid) return 1;
        else return 0;
    }

    /*
     * Calculate the quality of a solution with respect to this
     * cspInstance
     */
    public double quality(SolutionI solution)
    {
        // Cast the SolutionI object to a CSPSolution object
        CSPSolution cspSolution = (CSPSolution) solution;

        // Store the List of Var/Boolean pairs
        List<Entry<Var, Boolean>> s = cspSolution.getAssign().toList();
        // Store the List of Clauses from this CSPInstance
        List<Clause> list = getClauses();
        // Create a new reduced List
        List<Clause> reduced =
        s.fold(new List.Fold<Entry<Var,Boolean>,List<Clause>>()
        {
        public List<Clause> fold(Entry<Var,Boolean> a, List<Clause>
        p) { return reduce(p, a.getKey(), a.getVal()); }
        }

        // Pass the reduced List to a helper method for computation
        return satisfiedRatio(reduced);


    }

    /*
     * NOTE: For the purposes of providing a concise example, the
     * full implementation of the 'reduce' method, the
     * 'satisfiedRatio' method, and other additional helper methods
     * are not included in this documentation. However, the method
     * signatures are provided and the code functionality is explained
     * within comment blocks below. During implementation, it is
     * important to note that actual usable & valid Java code must be
     * provided for any helper methods that are called within the
     * required interface methods and that these helper methods
     * should be placed in the same class from which they are called.
```

```
 */

/* Reduce a given Problem based on a single Literal assignment */
public static List<Clause> reduce(List<Clause> clauses, final Var var,
final boolean val) { }

/* Calculate the satisfaction ratio of a (reduced) Problem */
public static double satisfiedRatio(List<Clause> clauses) { }

/* Accumulation for the satisfaction ratio */
static class R { }

}}

/*
 * Methods for the cspInstanceSet class.
 *
 * The cspInstanceSet class must implement the InstanceSetI
 * interface. Therefore, the ‘belongsTo’ method must be
 * implemented.
 */
cspInstance {{
// Check if an instance belongs to this cspInstanceSet
public Option<String> belongsTo(InstanceI instance)
{
	// Cast the InstanceI object to a CSPInstance object
	CSPInstance i = (CSPInstance) solution;
	// Verification variable
	boolean valid = true;
	// Iterator
	Iterator<Clause> clauses = i.getClauses().iterator();

	// For each clause in the Iterator...
	for(clauses; clauses.hasNext();)
	{
		// ...store the clause in a variable...
		Clause clause = clauses.next();
		/*
		 * see if this relation number in the clause
		 * is present in the CSPInstanceSet then valid else invalid
		 */
		valid &= getType().contains(clause.getRelnum());
	}

	// Return the result of the findings
	if(!valid)
```

```
            {
                  return new Some<String>(“The instance ” + i.print() +
                  ” is different from ” + this.print() + “.”);
            }
            return new None<String>();
      }

      }}

      /*
       * Customizable cspConfig parameters for the csp domain. These must
       * match the information defined in the cspConfig class (within the
       * cspDomain class dictionary).
       *
       * The cspConfig class must define DEFAULT_CSP_CONFIG, which specifies
       * the default configuration values for the cspConfig parameters
       * defined in the cspDomain class dictionary.
       *
       * The methods getDefaultDomainConfig() and getDefaultConfig() must
       * also be implemented, which return DEFAULT_CSP_CONFIG and a default
       * Config respectively.
       */
      cspConfig {{
                  private static cspConfig DEFAULT_CSP_CONFIG;
                  static {
                        try {
                              DEFAULT_csp_CONFIG = cspConfig.parse(

                              "csp_config[\n" +
                              "maxRelNum: 255\n" +
                              "maxVariables: 10\n" +
                              "]"
                              );
                        }

                  catch(Exception ex)
                  {
                         ex.printStackTrace();
                  }
            }

            public static cspConfig getDefaultDomainConfig()
            {
                  return cspConfig.DEFAULT_csp_CONFIG;
            }

            public static Config getDefaultConfig()
```

```
        {
              return new Config(SCGConfig.getDefaultSCGConfig(),
              getDefaultDomainConfig());
        }
    }}
```

3.3 Avatar Definitions

In addition to the domain specific class definitions and the domain behavior definitions, designers must provide an avatar definition that is specific to their domain. As previously mentioned, this avatar definition is used in the creation of a baseline avatar that is capable of playing the game without violating the rules. However, this avatar lacks any kind of sophisticated logic. Therefore, its functionality is mostly only useful for providing scholars with a solid framework from which they can implement better algorithmic strategies for competing in tournaments.

3.3.1 Overview

When a designer wants to create an avatar definition for their domain, they will need to define certain methods from the AvatarI interface (at the SCG level):

`List<Claim>` **`propose`**`(List<Claim> forbiddenClaims)`: The "propose" method is used to make new claims during competitions.

`List<OpposeAction>` **`oppose`**`(List<Claim> claimsToBeOpposed)`: The "oppose" method is used to respond to the claims of the proposer given in the input parameter `claimsToBeOpposed`.

`InstanceI` **`provide`**`(Claim claimToBeProvided)`: the "provide" method is used to provide an instance of the instanceSet for the given claim.

`SolutionI` **`solve`**`(SolveRequest solveRequest)`: the "solve" method is used to provide solution for the instance provided. This solution will be used to determine the outcome of the opposing action.

Additionally, the constructor for any avatar must include a `Config` object so that all regulations of the playground configuration are properly followed. Parameters of this object should be taken into account when designing the various methods described aboved.

These definitions are contained in a domain design specific avatar behavior file (i.e., "ddsAvatar.beh"). Additionally, a domain design specific avatar class dictionary (i.e., "ddsAvatar.cd") must be created so that the function definitions from the avatar behavior file can be automatically generated using DemeterF.

3.3.2 Code Samples

The code samples below represent a template class dictionary (i.e., "ddsAvatar.cd") and a template behavior definition (i.e., "ddsAvatar.beh") needed for a complete avatar definition. Within this code, comments (provided in green) explain the purpose of each line and/or what modifications are required when a domain designer is attempting to make changes.

References to domain design specific data are denoted in orange (i.e., dds). During actual implementation, this text should be replaced with a three letter representation that distinguishes a particular domain. For example, the Avatar class in the Highest Safe Rung (i.e., hsr) domain would be denoted by the "hsrAvatar" class. Therefore, a domain designer for the Highest Safe Rung problem would replace all "dds" text in the template below with "hsr". Note that file names should also reflect this change. Therefore, the resulting files in this example would be named "hsrAvatar.cd" (for the class dictionary) and "hsrAvatar.beh" (for the behavior file) if they were being created for the Highest Safe Rung problem.

```
/*
 * File: ddsAvatar.cd
 * (replace all instances of dds, including the file name, with
 * a three letter representation of the domain name)
 */

/*
 * "scg.cd" includes the class definitions for SCG courts.
 *
 * The SCG class dictionary has meta-level definitions
 * that specify how each avatar definition cd must be defined.
 */

nogen include "../scg/scg.cd";

/*
 * This avatar definition will be created within the
 * dds.Avatar package.
 */

package dds.Avatar;
```

```
/*
 * - Import the SCG package.
 * (necessary to ensure that this avatar is compatible with the
 * SCG courts system)
 * - Import the DDS package.
 * (necessary to ensure that this avatar can interact with
 * dds specific objects)
 *
 * After importing the SCG and DDS packages, you should also
 * import any additional packages or classes that will be
 * required for the implementation of methods in your .beh file
 */

import scg.*;
import dds.*;

// Enter additional 'import' statements before this comment.

/*
 * Define a domain design specific Avatar (i.e., ddsAvatar).
 *
 * The ddsAvatar definition must implement the AvatarI interface.
 * Therefore, AvatarI methods must be defined in your corresponding
 * behavior file (.beh) as part of the ddsAvatar class (see the
 * template avatar behavior definition for more details).
 */

ddsAvatar = implements AvatarI.
```

```
/*
 * File: ddsAvatar.beh
 * (replace all instances of dds, including the file name, with
 * a three letter representation of the domain name)
 */


/*
 * Methods for the ddsAvatar class.
 *
 * The ddsAvatar class must implement the AvatarI interface.
 * Therefore, the 'propose', 'oppose', 'provide', and 'solve'
 * methods must be implemented.
 */
ddsAvatar {{
/*
 * The constructor for the ddsAvatar class contains a Config
 * object.
 */
private Config config;

/*
 * The constructor to be called during registration (where you
 * supply a Config)
 */
public ddsAvatar(Config cfg)
{ config = cfg; }

/*
 * Proposes a List<Claim> that does not include any claims
 * from the given List<Claim> (i.e., forbidden claims)
 */
public List<Claim> propose(List<Claim> forbiddenClaims)
{
      // Replace with domain specific logic:
      return null; // Must return a 'List<Claim>' object
}

// Decides what opposition action to take for each claim in the given
List<Claim>.
public List<OpposeAction> oppose(List<Claim> claimsToBeOpposed)
{
      // Replace with domain specific logic:
      return null; // Must return a 'List<OpposeAction>' object
}

// Provides a ddsInstance for the given Claim
public InstanceI provide(Claim claimToBeProvided)
{
      // Replace with domain specific logic:
      return null; // Must return a 'ddsInstance' object
}

// Solves (i.e., gives a ddsSolution) for the instance in the given
SolveRequest
public SolutionI solve(SolveRequest solveRequest)
{

       ddsInstance i = (ddsInstance)solveRequest.getInstance();
      // Replace with domain specific logic:
      return null; // Must return a 'ddsSolution' object
}

/*
 * Include helper methods for the 'propose', 'oppose', 'provide',
 * and/or 'solve' methods here (i.e., before the double curly
 * brackets)
 */

}}
```

3.3.3 Example

The example provided below describes an avatar definition for the **c**onstraint **s**atisfaction **p**roblem (i.e., "CSP"). Therefore, this avatar definition consists of two files - a class dictionary file named "cspAvatar.cd" and a behavior definition file named "cspAvatar.beh". Comments (provided in green) explain the changes (provided in light blue). While these comments are unnecessary when actually creating an avatar definition, they are provided in this document to highlight what modifications were made from the original template (in Section 3.3.2). Additional comments (in black within the behavior definition) explain the implemented code functionality.

```
        // File: cspAvatar.cd

        // This line is the same in all avatar definitions:
        nogen include "../scg/scg.cd";

        // Changed package name from 'dds.Avatar' to 'csp.Avatar'
        package csp.Avatar;

        // This line is the same in all avatar definitions:
        import scg.*;
        // Changed package name from 'dds.*' to 'csp.*'
        import csp.*;
        // Added import of the Random class (for use in cspAvatar.beh):
        import java.util.Random;

        // Changed from 'ddsAvatar' to 'cspAvatar'
        cspAvatar = implements AvatarI.
```

```
        // File: cspAvatar.beh
```

```
/*
 * Methods for the cspAvatar class.
 *
 * The cspAvatar class must implement the AvatarI interface.
 * Therefore, the 'propose', 'oppose', 'provide', and 'solve'
 * methods must be implemented.
 */
cspAvatar {{

/*
 * The constructor for the cspAvatar class contains a Config
 * object.
 */
private Config config;

/*
 * The constructor to be called during registration (where you
 * supply a Config)
 */
public cspAvatar(Config cfg)
{ config = cfg; }

/*
 * Proposing random claims which are not in the given List<Claim>
 * (i.e., forbidden claims)
 *
 * These random claims are added to a List<Claim>, which is
 * eventually the object that is returned.
 */
public List<Claim> propose(List<Claim> forbiddenClaims)
{
      // Start with an empty List<Claim> object
      List<Claim> claims = List.create();

      // Do the following x number of times (0 < x < max number of proposals)...
      for(int i=0; i < config.getScgCfg().getMaxProposals(); i++)
      {
            // ...get a random claim (from the 'generateRandomClaim' method)...
            Claim claim = generateRandomClaim();

            /*
             * ...make sure it is unique (i.e., not in the
             * forbidden list or in our list already)...
             */
            while(forbiddenClaims.contains(claim) &&
                  claims.contains(claim) )
```

```
                { claim = generateRandomClaim(); }

                // ...and add it to the list to eventually be proposed
                claim = claims.append(claim);
        }

        // Return the resulting List<Claim>
        return claims;
}

// Randomly decides which claims from the given List<Claim> to oppose
public List<OpposeAction> oppose(List<Claim> claimsToBeOpposed)
{
                return claimsToBeOpposed.map(new List.Map<Claim,
                                           OpposeAction>() {

                        public OpposeAction map(Claim claim)
                        {
                                Random rand = new Random();

                                int randOppose = rand.nextInt(3);

                                if(randOppose == 0)
                                        return new Agreement();

                                else if(randOppose == 1)
                                {

                                        double q = claim.getQuality();
                                        SCGConfig scg_cfg = config.getScgCfg();

                                        if (claim.getProtocol() instanceof PositiveSecret)
                                        {
                                                if(q > scg_cfg.getMinStrengthening())

                                                return new Strengthening(
                                                claim.getQuality() -
                                                scg_cfg.getMinStrengthening());

                                                else return new Refuting();

                                        }
                                        else
                                        {

                                                if(q < scg_cfg.getMinStrengthening())
                                                {
                                                        return new
                                                        Strengthening(claim.getQuality() +
                                                scg_cfg.getMinStrengthening());
                                          }
                                          else return new Refuting();
                                    }

                              }

                        }
                        else return new Refuting();
                  });
      }

      // Provides a symmetric Instance of the given instanceSet
      public InstanceI provide(Claim claimToBeProvided)
      {
                  CSPInstanceSet cspInstanceSet = (CSPInstanceSet)
                  claimToBeProvided.getInstanceSet();
                  CSPConfig cfg = (CSPConfig) config.getDomainConfig();
                  int numVars = cfg.getMaxVariables();
                  List<Var> vars = List.<Var>create();

                  for(int i = 1; i <= numVars;i++)
                  {
                        vars = vars.append(new Var(new ident("v" + i)));
                  }

                  CSPInstance cspInstance = new CSPInstance(vars,
                  symmetric(cspInstanceSet.getType().toList(), numVars, 3));

                  return cspInstance;

      }

      // Solve using a random assignment of values to all the variables
      public SolutionI solve(SolveRequest solveRequest)
      {
                  CSPInstance i = (CSPInstance)solveRequest.getInstance();

                  Random rand = new Random();

                  CSPSolution solution = new CSPSolution(new ListMap<Var,
                  Boolean>(randomAssign(i.getVars(), rand.nextDouble())));

                  return solution;
```

```
    }

    /*
     * NOTE: For the purposes of providing a concise example, the
     * full implementation of the ‘generateRandomClaim’ method, the
     * ‘generateRandomAllowedProtocol’ method, and other additional
     * helper methods are not included in this documentation. However,
     * the method signatures are provided and the code functionality is
     * explained within comments block below. During implementation,
     * it is important to note that actual usable & valid Java code must
     * be provided for any helper methods that are called within the
     * required interface methods and that these helper methods
     * should be placed in this class.
     */

    Private Claim generateRandomClaim() { }

    /* Generates a random protocol instance from the given protocolsAllowed */
    private ProtocolI generateRandomAllowedProtocol(Cons<FullyQualifiedClassName>
    protocolsAllowed) { }

    /* Create a symmetric formula using the given Relation numbers */
    public static Cons<csp.Clause> symmetric(List<Integer> rs, int numVars, int rank) { }

    /* Sort the relations to find the most important one... */
    public static int importantRelation(CSPInstanceSet t) { }

    /* Determine the number of bits set in the given relation number */
    public static int bitsSet(int i) { }

    /* Push the given number onto the front of each list */
    private static class PushN<X> extends List.Map<List<X>,List<X>> { }

    }}
```

4 Protocol Definitions

(NOTE: For the purposes of this documentation, always assume that '*Alice*' is the person making the claim and that '*Bob*' is the person acting on / opposing the claim.)

A protocol includes a sequence of steps involved in refuting a claim and a predicate that determines the outcome of the refutation. Strengthening and agreement of a claim use the defined steps in refutation protocol. Each step in the protocol involves Alice or Bob providing data, which includes instances and solutions. Once all the steps are completed, the predicate evaluates the outcome of the refutation using these instances and solutions.

When playground designers are creating new configurations, they can use the protocols that already exist in the SCG court system. If a domain demands a sequence of steps and an evaluation predicate that cannot be satisfied by the existing protocols, however, then a new protocol must be defined.

The following sections [1] illustrate a portion of the SCG class dictionary that indicates how to specify the protocol steps, [2] provide a template java class that can be used as a baseline when creating new protocols, and [3] describe an example of a protocol called `NegativeSecret`.

4.1 Overview & Protocol Steps

When a protocol designer defines a new protocol, it is mandatory that they define a protocol for both positive claims and negative claims. This is a condition imposed by the SCG court system as the protocol for a negated claim is useful in cases of agreement (see Section 2.2.3).

A protocol is a Java class that implements the `ProtocolI` interface. Therefore, the following 2 methods must be defined:

> `double` **`getResult`**`(Claim claim, SolutionI[] solutions, InstanceI[] instances)`: A predicate that determines that outcome of the refutation.
>
> `ProtocolSpec` **`getProtocolSpec`**`()`: A method to fetch the sequence of steps specified in the protocol.

As mentioned, the `ProtocolSpec` class is defined within the SCG class dictionary:

```
/* From "SCG.cd" */

ProtocolSpec = <steps> List(Step).

Step = <action> Action "from" <role> Role.
```

```
interface Role = Alice | Bob.

Alice = "Alice".
Bob = "Bob".

interface Action = ProvideAction | SolveAction.

// Can only provide for the claim

ProvideAction = "instance".

/*
 * solve the instance provided in step # stepNo
 * step # -1 for the singleton instance in claim
 * stepNo is 0-based
 */

SolveAction = "solution" "of" *s <stepNo> int.
```

An example of the protocol steps that confers to this class dictionary is given below:

```
instance from Alice

solution of 0 from Alice

solution of 0 from Bob
```

This protocol has the following steps:

· The player playing the role of *Alice* has to provide an instance in the first step.
· The player playing the role of *Alice* has to provide a solution for the instance in the first step.
· The player playing the role of *Bob* has to provide a solution for the instance in the first step.

4.2 Code Sample

The code sample below is a template class (i.e., "`Protocol.java`") that protocol designers can modify to specify their new protocol. Within this code, comments (provided in green) explain the purpose of each line and/or what modifications are required when a protocol designer is attempting to make changes. Text that should be changed to reflect a specific protocol name (i.e., your new file name) is denoted in orange. During actual implementation, this text should be replaced with the name of a particular protocol. For example, if your result protocol is `ReverseProtocol.java`, all text is orange should be changed to “ReverseProtocol” as well. Text in blue should be replaced with protocol specific code.

```
package scg.protocol;

import scg.*;

public class protocol implements ProtocolI
{
      private static ProtocolSpec PROTOCOL_SPEC;
      static {
            try {
            PROTOCOL_SPEC = ProtocolSpec.parse (
            /* Sequence of steps involving Alice/Bob
             * providing instance/solution.
             * (see Section 4.1 to define this)
             */
             [...]
            );
      }
      catch(Exception ex)
      {
            ex.printStackTrace();
      }
}

/*
 * This method always returns a real number between -1 and 1.
 *
 * If this number is negative, it indicates that the refutation
 * is successful and Bob wins.
 * If this number is positive, it indicates that refutation fails
 * and Alice successfully defends
 */
public double getResult(Claim claim, SolutionI[] solutions,
                        InstanceI[] instances) {
       /*
        * Code that evaluates the quality of solutions for
        * the instances should be written here.
```

```
              *
              * The quality of the claim can also be considered
              * for this evaluation.
              */
            [...]
       }

       public ProtocolSpec getProtocolSpec()
       {
             return protocol.PROTOCOL_SPEC;
       }

       private static protocol instance = new protocol();

       public static protocol getInstance()
       {
             return instance;
       }
}
```

4.3 Example

The example provided below describes the `NegativeSecret` protocol. Comments (provided in green) explain the changes (provided in light blue). These comments should be removed from the file when actually creating the protocol class and should be replaced with comments that explain the specific functionality that the protocol designer implements. They are provided in this document to highlight what modifications were made from the original template (in Section 4.2).

```
         // File: protocol.java

         package scg.protocol;

         import scg.*;

         /* Replaced protocol with NegativeSecret */
         public class NegativeSecret implements ProtocolI {

            private static ProtocolSpec PROTOCOL_SPEC;
            static {
                  try {
                  PROTOCOL_SPEC = ProtocolSpec.parse (
                  /* Replaced [...] with protocol specific steps */
                  "instance from Alice " +
                  "solution of 0 from Alice " +
                  "solution of 0 from Bob "
                  );
            }
            catch(Exception ex)
            {
                  ex.printStackTrace();
            }
      }

     /*
      * Replaced [...] with code that determines the result using
      * the instances and solution specified in the protocol steps.
      *
      * The code in blue evaluates the solution given by Alice and
      * Bob for the instance provided by Alice.
      * It compares the quality of the 2 solutions and
      * also uses claim quality in the comparison to evaluate the
      * outcome of the refutation.
      */
      public double getResult(Claim claim, SolutionI[] solutions,
                  InstanceI[] instances) {
                        if(solutions.length == 2 && instances.length == 1)
                        {
                              InstanceI i = instances[0];
                              SolutionI aliceSolution = solutions[0];
                              SolutionI bobSolution = solutions[1];

                              if(i.quality(bobSolution) >=
                                (i.quality(aliceSolution) *
                                claim.getQuality()))
                                    return -1; // Bob win
                              else return 1; // Alice win
                        }
                        return 0; // Draw
      }

      public ProtocolSpec getProtocolSpec()
      {
            return NegativeSecret.PROTOCOL_SPEC;
      }

      private static NegativeSecret instance = new NegativeSecret();

      public static NegativeSecret getInstance()
      {
            return instance;
      }
}
```

5 Playground Configuration

(NOTE: In the current implementation, playground configurations are restricted to utilizing only domains and protocols that are natively included as part of the SCG court system. Therefore, it is not possible to use newly created domains and protocol when creating a tournament. In future implementations, it should be possible to submit non-native domains and protocols simultaneously with a playground configuration or update the SCG court system with new domains and protocols beforehand via an external interface).

After a domain has been fully defined and a protocol has been selected, they can be used as part of a playground configuration. A playground configuration defines the constraints for a particular tournament. While this obviously includes the specification of which domain and protocol(s) to use, it also includes setting generic parameters that apply to any scientific community game.

The playground configuration consists of 2 parts:

`SCGConfig`: configuration that defines the domain, protocol, and general SCG parameters that apply to all scientific community games.

`DDSConfig`: configuration that specifies custom parameters that are explicit to a domain (NOTE: The actual name of this class will be defined within the class dictionary for the domain specified in the `SCGConfig` portion of the playground configuration)

The following sections [1] describe why each parameter within a playground configuration is important, [2] illustrate a blank playground configuration template, and [3] provide an example of a playground configuration.

5.1 SCG Configuration

The SCG configuration class (i.e., "`SCGConfig`") is defined within the SCG level class dictionary (i.e., "`scg.cd`").

The parameters in the SCG configuration are:

- `domain`: fully qualified class name of the domain to be used for the game
- `protocols`: fully qualified class names of the protocols that can be used in the game
- `tournamentStyle`: either 'full round-robin', 'knockout', or 'swiss'
- `turnDuration`: an integer indicating the number of seconds allowed for each avatar's turn

- `maxNumAvatars`: an integer indicating the maximum number of avatars allowed for this tournament
- `minStrengthening`: a double indicating the minimum value by which a claim should be strengthened
- `initialReputation`: an integer indicating the initial reputation of the avatars when the tournament begins
- `maxReputation`: a double indicating the maximum reputation that can be attained by the avatars
- `reputationFactor`: a double in [0,1] used to determine the reputation gain upon successful defense of a claim or upon successful refutation of the claim of another scholar
- `minProposals`: an integer indicating the minimum number of proposal that must be in a response from an avatar
- `numRounds`: an integer indicating the number of rounds to be played in the game
- `proposedClaimMustBeNew`: a Boolean indicating whether proposed claim must be different from previously proposed claims
- `minConfidence`: a double in [0,1] that specifies the minimum amount of confidence that an avatar can have for their claim

5.2 Domain Configuration

The domain configuration specifies the constraints specific to a given domain. The domain configuration class c is defined in the class dictionary for the domain that is specified in the SCG configuration (though it is the class that implements that `DomainConfigI` interface and is not necessarily named `DDSConfig`). The example in section 5.4 shows how domain level constraints are specified for the CSP game (as previously defined in section 3.1.3).

5.3 Code Sample

The code sample below is a template playground configuration (i.e., “`PGConfig.txt`”) that playground designers can modify to customize the existing game domains in SCG that satisfy their needs.

References to domain specific data are denoted in orange (i.e., dds). During actual implementation, this text should be replaced with a three letter representation that distinguishes a particular domain.

All parameters within `scg_config[...]` are SCG configurations. All parameters within `dds_config[...]` are domain configurations.

All references to `[...]` in the scg_config should be replaced by suitable values (see Section 5.1). The reference to `[...]` in the dds_config should be replaced by suitable parameter values which are specified in the `ddsConfig` class in the domain class dictionary (see Sections 3.1.1 and 3.1.2).

```
scg_config[
domain:[...]
protocols: [...]
tournamentStyle: [...]
turnDuration: [...]
maxNumAvatars: [...]
minStrengthening: [...]
initialReputation: [...]
maxReputation: [...]
reputationFactor: [...]
minProposals: [...]
maxProposals: [...]
numRounds: [...]
proposedClaimMustBeNew: [...]
minConfidence: [...]
]

/*
 * Replace the blue text below with the fully qualified class name
 * of a class that implements the DomainConfigI in ddsDomain.cd
 * (see Sections 3.1.1 and 3.1.2 for more details)
 */
[fully qualified class name of a Domain config class] {{ dds_config[
[...]
]
}}
```

5.4 Example

The example provided below describes a possible configuration for the **c**onstraint **s**atisfaction **p**roblem (i.e., "CSP"). Comments (provided in green) explain the changes (provided in light blue). These comments must be removed from the file when actually utilizing the playground configuration file. However, they are provided in this document to highlight what modifications were made from the original template (in Section 5.3).

```
scg_config[
domain:csp.CSPDomain
protocols: scg.protocol.NegativeSecret scg.protocol.PositiveSecret
tournamentStyle: full round-robin
turnDuration: 60
maxNumAvatars: 20
minStrengthening: 0.01
initialReputation: 100.0
maxReputation: 1000.0
reputationFactor: 0.4
minProposals: 2
maxProposals: 5
numRounds: 9
proposedClaimMustBeNew: true
minConfidence: 0.01
]

/*
 * Modified to specify the parameters of the CSPConfig class
 * (see Section 3.1.3 for the full definition of this class)
 */
csp.CSPConfig {{ csp_config[
maxRelNum: 255
maxVariables: 10
]
}}
```

6 Managing Tournaments & Users

(NOTE: Images shown in the subsequent sections have been slightly aesthetically modified for document integration. Furthermore, the address in the images points to *localhost*. Consult your system administrator for the address of your specific SCG court system.)

The SCG court system manages tournament initialization and provides relevant data via a simple web interface. Through this, administrators can create new tournaments that approved users can enroll in and eventually register for. If a user is not yet approved, they can also sign up for an account and request access, which is controlled by the administrator. The following sections detail the various levels of functionality contained in the aforementioned interface and outline the standard chronological procedure for operating a successful tournament.

6.1 Sign In

The sign in page is the primary entry point to the web interface. From here, administrators can log in and access the admin control panel (see Section 6.2). Similarly, regular users can also log in and access any tournaments in the system using this page. If you don't have an account, you can sign up for one via the "Sign Up" link at the bottom. However, signing in with that account will be prohibited until the administrator approves it.

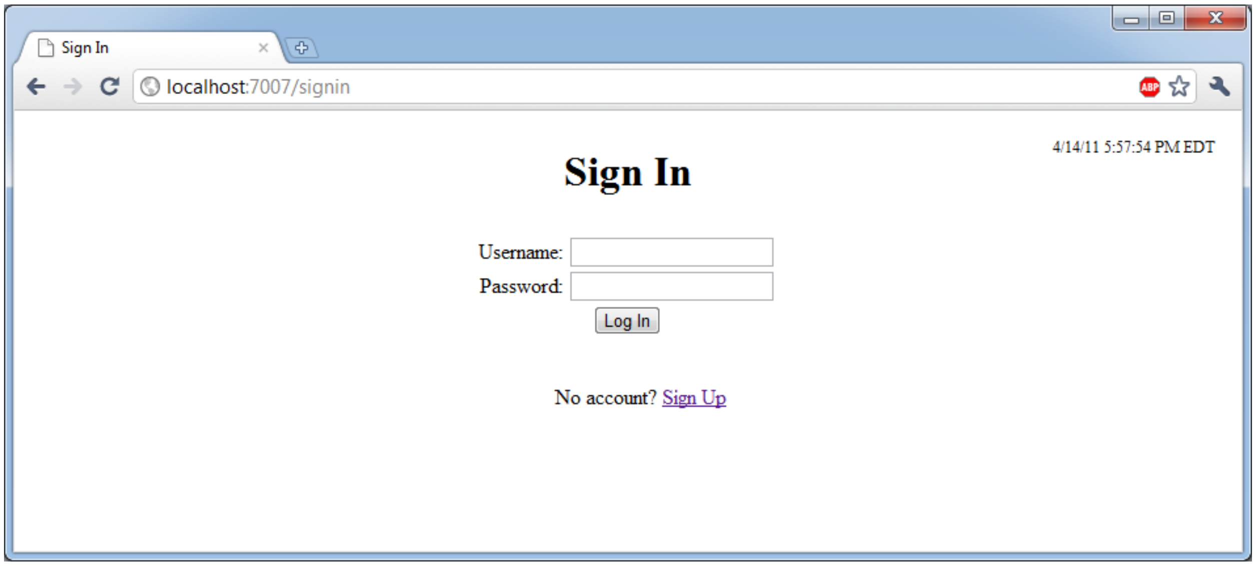
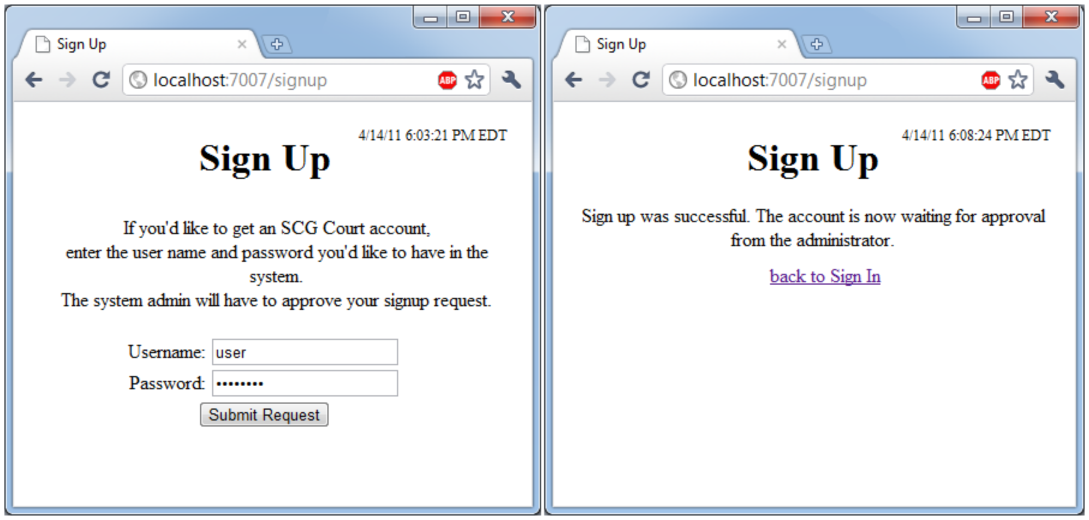


Figure A : The Sign In Page

6.1.1 Sign Up

As mentioned, potential users can sign up for an account by clicking the appropriate link from the sign in page. Once they have submitted their request, they will be notified that the process was successful and that the account is awaiting approval. The user will be able to sign in after the administrator approves the account with the credentials that were used in during the sign up process.



If you'd like to get an SCG Court account,
enter the user name and password you'd like to have in the system.
The system admin will have to approve your signup request.

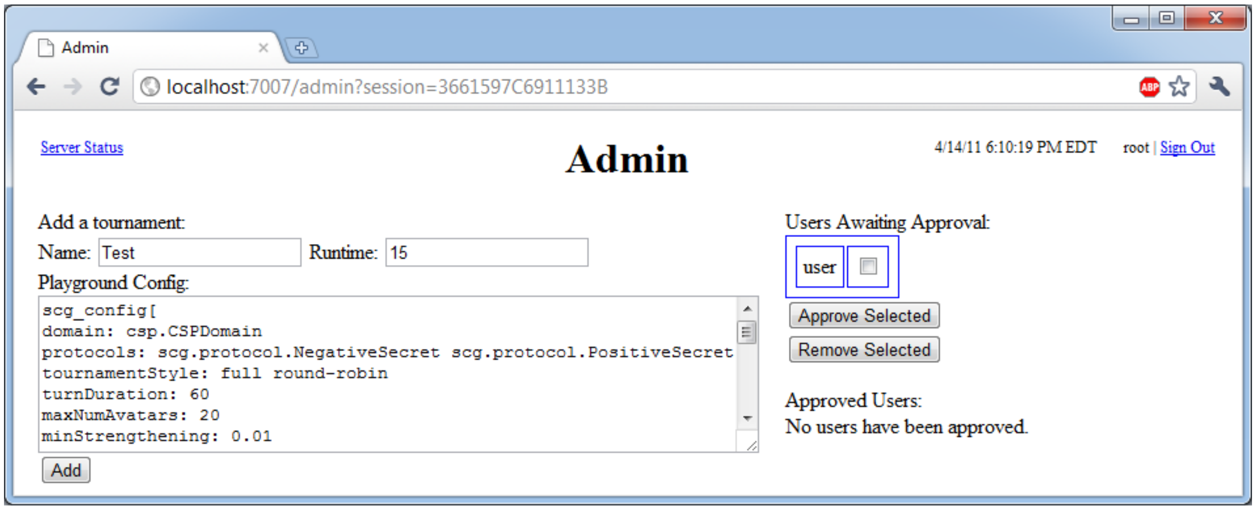
Username: user
Password: ••••••••

Sign up was successful. The account is now waiting for approval from the administrator.

Figure B: Signing Up

6.2 Admin Control Panel

The Admin Control Panel enables the admin to control and maintain the tournaments, users of SCG court. Access to the admin control panel can be obtained by entering the proper credentials at the sign in page. Once the username and password is validated, a page will appear with options to [1] add a tournament, [2] approve or remove users, and [3] view the server status (i.e., manage existing tournaments). Adding a tournament and approving/removing users are both relatively simple processes whereas viewing the server status yields more complex functionality.



Add a tournament:
Name: Test Runtime: 15
Playground Config:

```
scg_config[
domain: csp.CSPDomain
protocols: scg.protocol.NegativeSecret scg.protocol.PositiveSecret
tournamentStyle: full round-robin
turnDuration: 60
maxNumAvatars: 20
minStrengthening: 0.01
```

Approved Users:
No users have been approved.

Figure C: The Admin Control Panel

6.2.1 Adding Tournaments

To add a tournament, the administrator must provide 3 pieces of data:

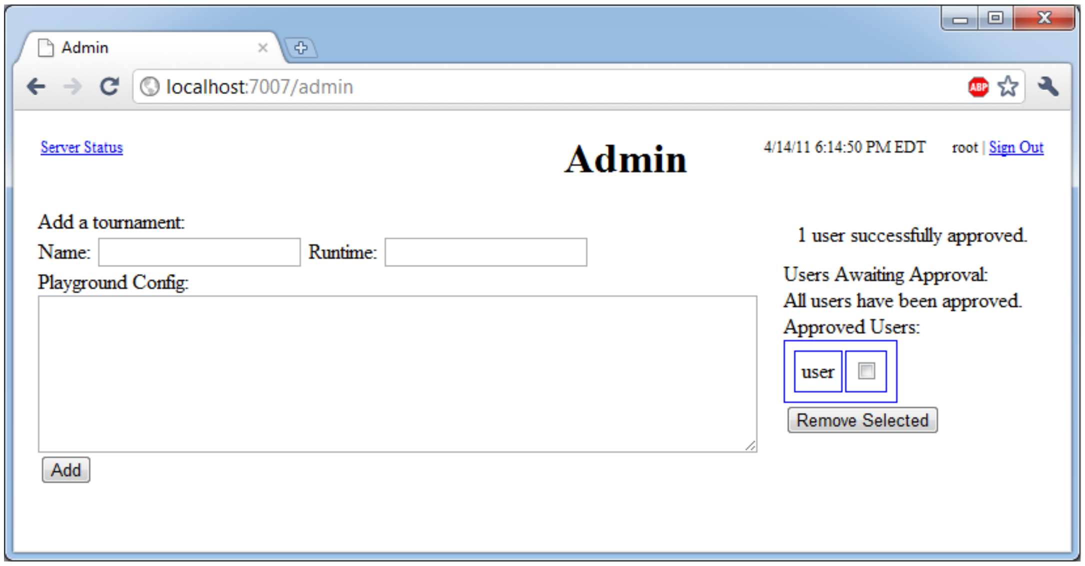
- `Name:` An arbitrary title used to identify the tournament on the server.
- `Runtime:` The number of minutes before the tournament will start.
- `Playground Config:` A valid playground configuration object (as described in section 5).

After providing this information and clicking the 'Add' button, the web interface will notify the administrator if there were errors in creating the tournament or if it has been scheduled successfully.

The example shown in the images creates a tournament named "Test" that starts in 15 minutes using the playground configuration from section 5.4.

6.2.2 Approving/Removing Users

Besides adding tournaments, the administrator can approve or remove users directly from the admin control panel. After logging in, pending users (i.e., “users awaiting approval”) will be shown on the right (see *Figure C*). Additionally, the administrator can elect to remove users that had previously been approved (see *Figure D*).



Admin

4/14/11 6:14:50 PM EDT root | Sign Out

Add a tournament:
Name: Runtime:
Playground Config:

1 user successfully approved.

Users Awaiting Approval:
All users have been approved.
Approved Users:

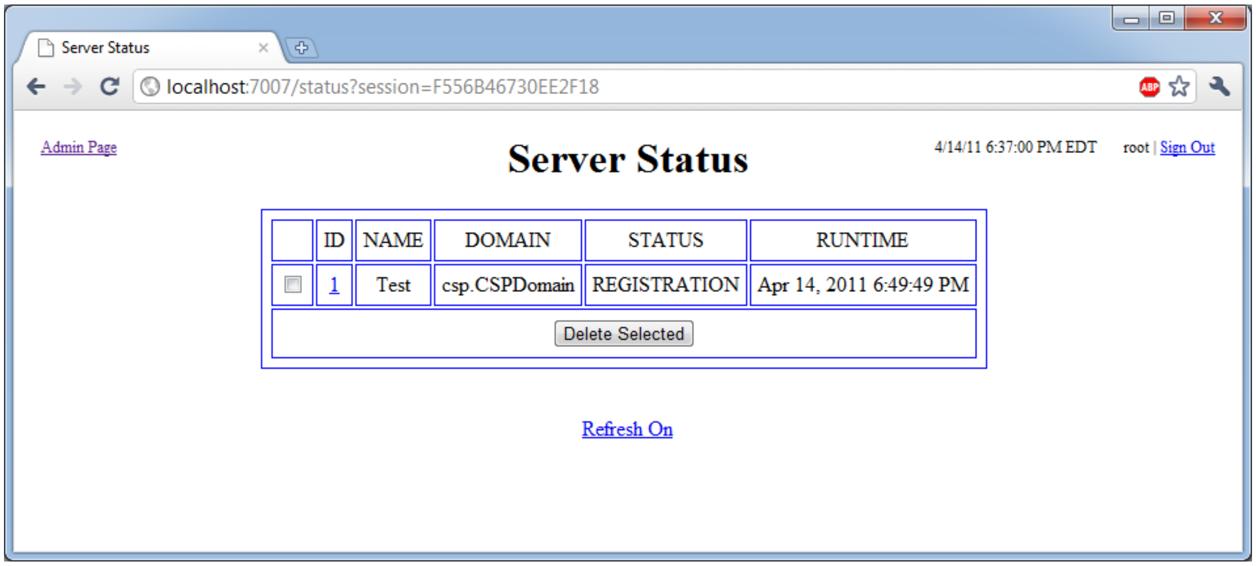
user	☐

Figure D: The Admin Control Panel (after approving a user)

6.3 Server Status Page

Once there is a sufficiently large group of approved users and at least one tournament has been created, it will be useful for the administrator to access the server status page to monitor any existing tournaments. By clicking the "Server Status" link at the top left of the admin control panel, the administrator will be brought to a page with a list of existing tournaments, which includes relevant information such as tournament ID number, tournament name, and tournament status. Additionally, tournaments that are no longer needed can be deleted from this page by clicking the checkbox in the corresponding row and selecting the 'Deleted Selected' button.

As a regular user, the server status page is the first accessible page after logging in. These users will also be able to view all the necessary details pertaining to existing tournaments. However, they will obviously be unable to navigate to the admin control panel or delete tournaments from this page.



Server Status

	ID	NAME	DOMAIN	STATUS	RUNTIME
☐	1	Test	csp.CSPDomain	REGISTRATION	Apr 14, 2011 6:49:49 PM

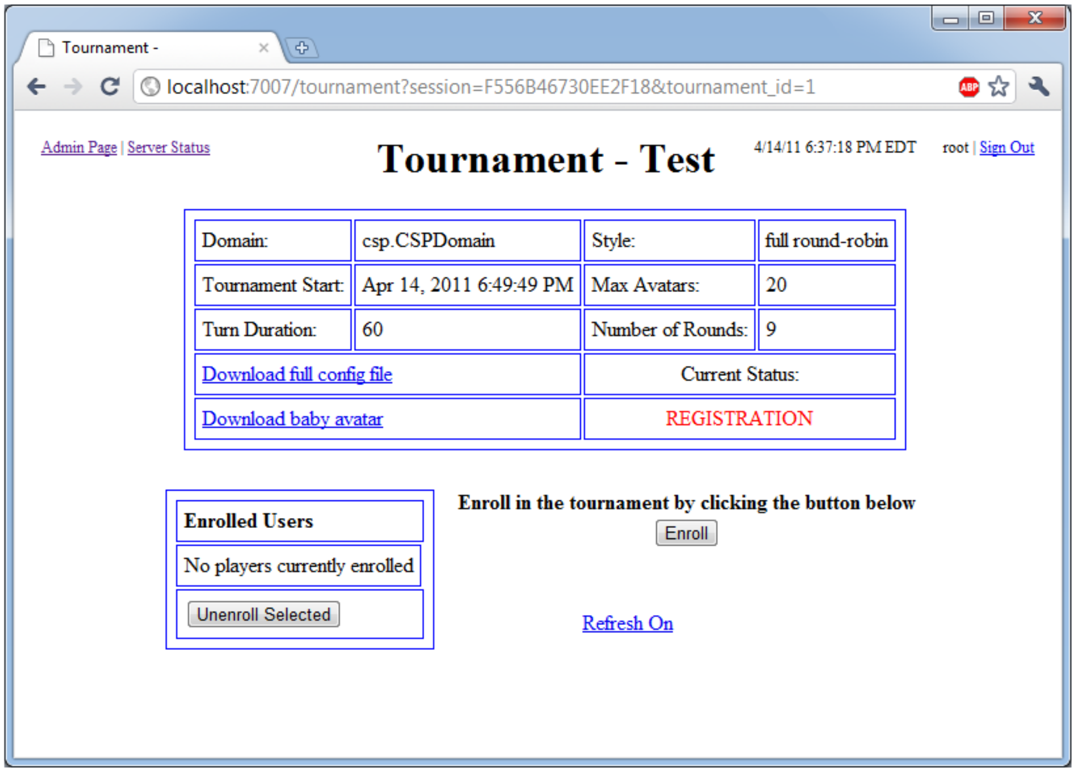
Figure E: The Server Status Page (Administrator View)

6.4 Tournament Status Page

From the server status page, clicking on a tournament ID number will navigate to that tournament's status page. From here, more details regarding a specific tournament are displayed (i.e., the tournament style, the maximum number of avatars, turn duration, and more). Links to download the playground configuration and the baby avatar are also available here.

Like the server status page, individual tournament status pages are also available to both the administrator and regular users. Regular users must sign in and open a particular tournament's status page to enroll in that tournament before registering. While regular users can also unenroll themselves from tournaments that they have previously enrolled in, only the administrator can unenroll any user. The unenroll option will only be available before the tournament is running.

Once the tournament has started (i.e., the current status is RUNNING), users can view the live score updates from this page. The "Refresh On" option automatically refreshes the page after certain time interval to get the most recent scores. After the tournament has completed, the current status changes to COMPLETED and the final rankings of the avatars will be displayed.


Tournament -
localhost:7007/tournament?session=F556B46730EE2F18&tournament_id=1
Admin Page | Server Status
4/14/11 6:37:18 PM EDT
root | Sign Out
Enroll
Unenroll Selected
Refresh On

Tournament - Test

Domain:	csp.CSPDomain	Style:	full round-robin
Tournament Start:	Apr 14, 2011 6:49:49 PM	Max Avatars:	20
Turn Duration:	60	Number of Rounds:	9
Download full config file		Current Status:	
Download baby avatar		REGISTRATION	

Enrolled Users
No players currently enrolled

Enroll in the tournament by clicking the button below

Figure F: A Tournament Status Page (Administrator View)

6.5 Tournament Setup

Given the explanations of the various components from the previous subsections, using the SCG court system's web interface to create and ultimately run a tournament should be a relatively easy procedure:

Prerequisites

- A sufficient number of users have successfully created accounts via "Sign Up" page (see Section 6.1.1).
- The administrator has successfully accessed the admin control panel (see Section 6.2).
- From the admin control panel, the administrator has successfully created a tournament (see Section 6.2.1).
- Also from the admin control panel, the administrator has successfully approved every user that has created an account in the first prerequisite (see Section 6.2.2).

Steps

- Approved users can successfully sign in via the SCG court system's web interface (see Section 6.1).
- A logged in user is able to observe the server status page, which contains the tournament that the administrator has previously created (see Section 6.3).
- By clicking on the appropriate tournament's ID number, the user is brought to the tournament status page and is able to enroll in the tournament (see Section 6.4).

Once tournament setup is complete, no additional steps are needed on the web interface side. However, users that are enrolled will need to register their avatars at some point once the tournament status has moved from "ENROLLEMENT" to "REGISTRATION". While users can still enroll during this time, they will not be able to register until this change (which takes place approximately 90 minutes before the tournament is scheduled to start).

6.5.1 Registration

The final step in playing a tournament is to register the avatars with the SCG court.

Prerequisites

- The user is successfully enrolled for the tournament and is able to download the config and baby avatar files from the tournament status page (see Section 6.4).
- The tournament's current status is "REGISTRATION".

Steps

- Run the following command from the console:

```
java scg.net.avatar.PlayerMain <portNo> < hostname > <username>
<password> <tournamentID>
```

`<portNo>` : port number listed on the tournament status page after enrolling

`<hostname>` : hostname where the SCG court system is running

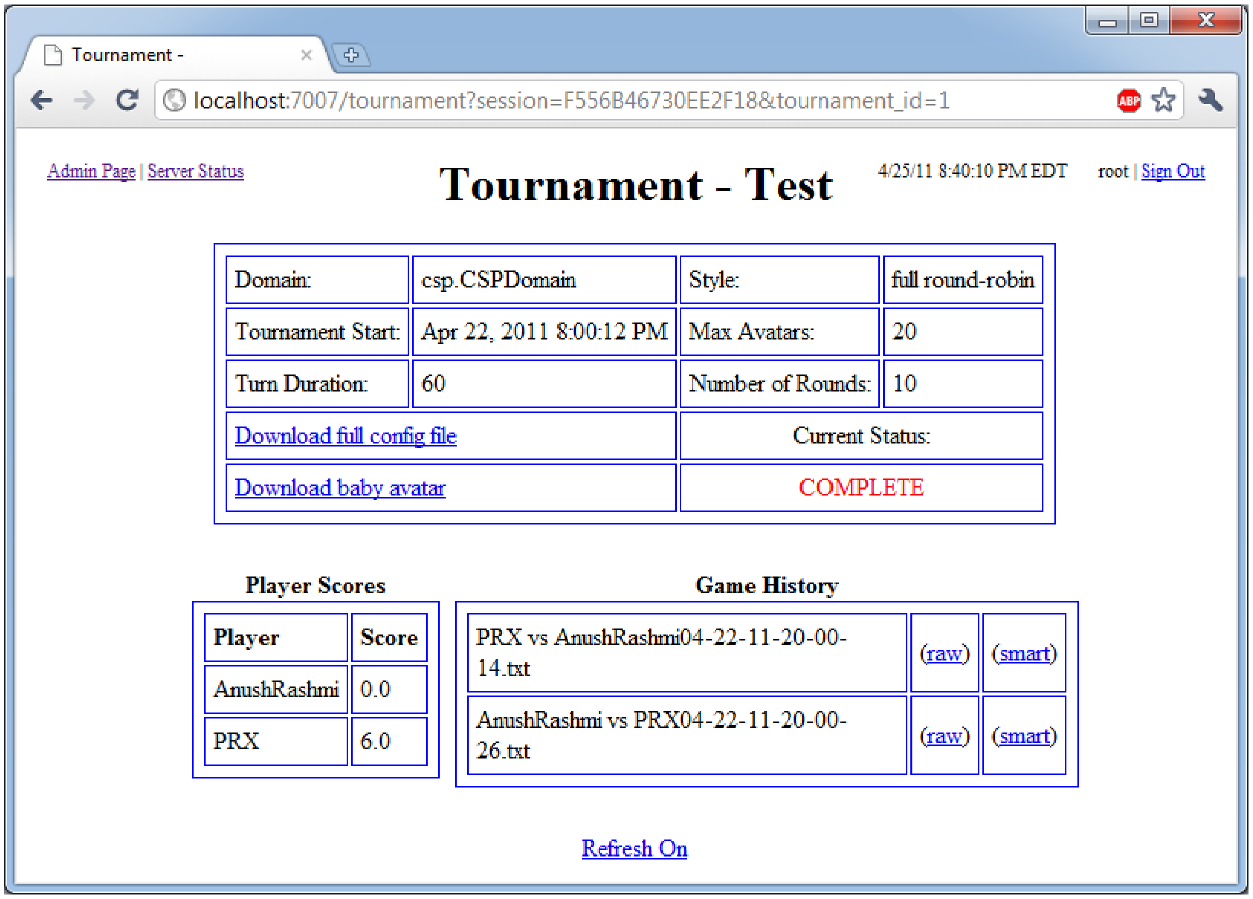
`<username>` : username that was used for enrolling in this tournament via the web interface

`<password>` : password for the username provided in the previous step

`<tournamentID>` : tournament ID mentioned on the server status page

After running the following command, the console should display the success message to indicate that registration of the avatar is complete. It is suggested that GNU screen is used to when running the command to ensure that connection errors do not occur before the tournament begins.

7 Data Mining / Histories


Tournament -

localhost:7007/tournament?session=F556B46730EE2F18&tournament_id=1

Admin Page | Server Status

Tournament - Test

4/25/11 8:40:10 PM EDT root | Sign Out

Domain:	csp.CSPDomain	Style:	full round-robin
Tournament Start:	Apr 22, 2011 8:00:12 PM	Max Avatars:	20
Turn Duration:	60	Number of Rounds:	10
Download full config file		Current Status:	
Download baby avatar		COMPLETE	

Player Scores

Player	Score
AnushRashmi	0.0
PRX	6.0

Game History

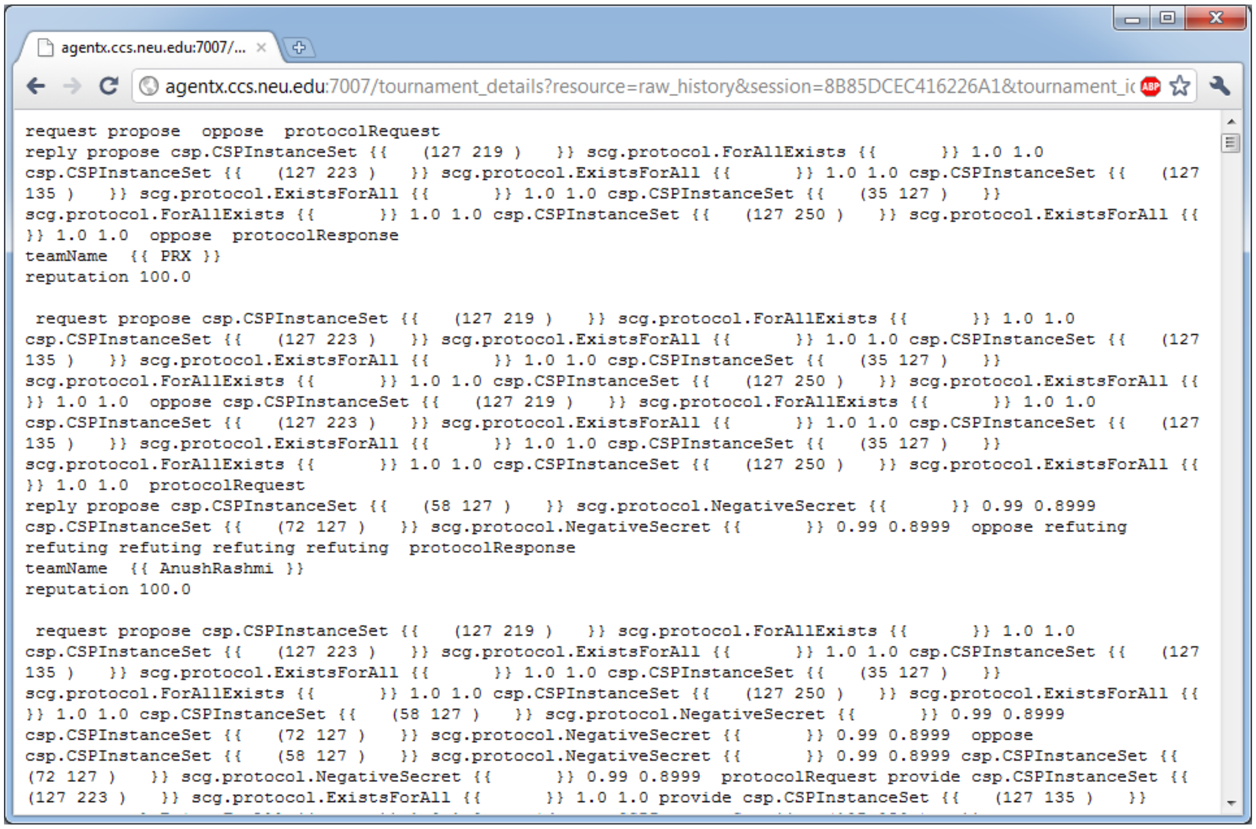
PRX vs AnushRashmi04-22-11-20-00-14.txt	(raw)	(smart)
AnushRashmi vs PRX04-22-11-20-00-26.txt	(raw)	(smart)

Refresh On

Figure G: A Tournament Status Page of a completed tournament

Once a tournament is complete (i.e., the current status is COMPLETE), users can access that tournament's status page to view both the raw and smart history files. The information contained in these files will vary based on the playground configuration. In general, however, they will provide relevant information regarding what actions took place during the competition. This feature is useful for scholars (i.e., avatar developers) as they will be able to see what types of flaws hinder their coding logic.

7.1 Raw History

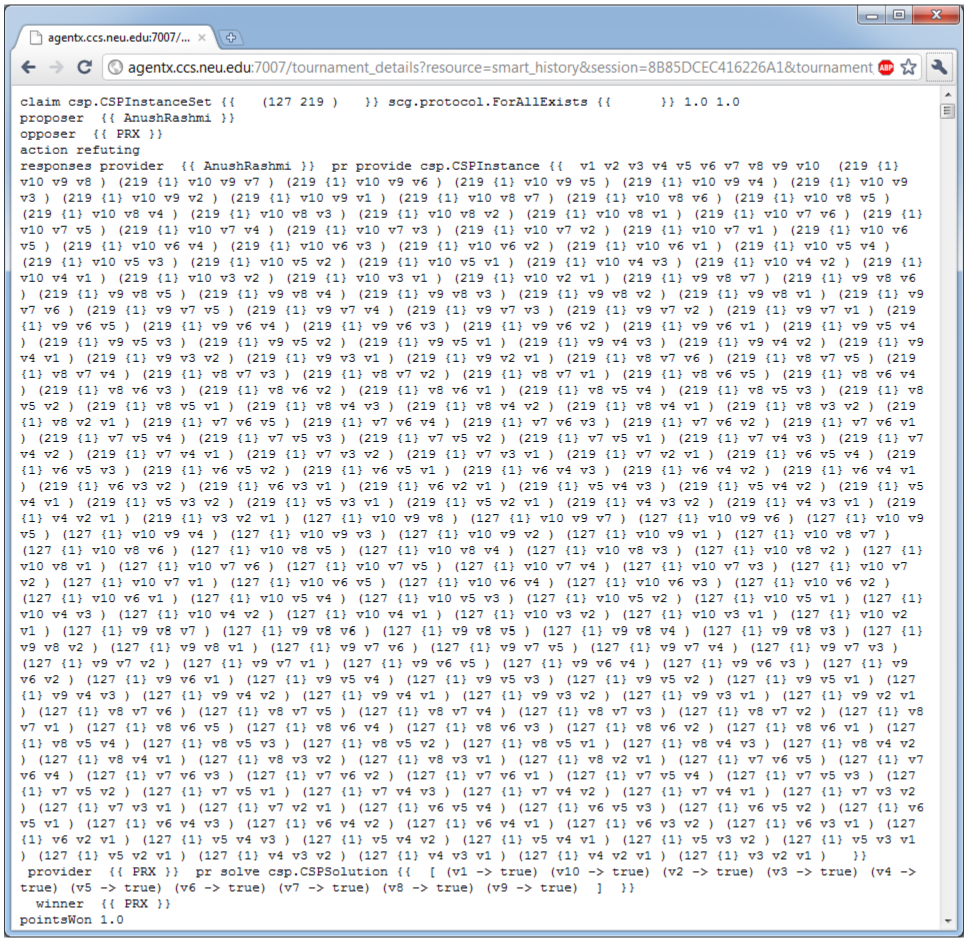

agentx.ccs.neu.edu:7007/...
agentx.ccs.neu.edu:7007/tournament_details?resource=raw_history&session=8B85DCEC416226A1&tournament_i

```
request propose  oppose  protocolRequest
reply propose csp.CSPInstanceSet {{   (127 219 )   }} scg.protocol.ForAllExists {{      }} 1.0 1.0
csp.CSPInstanceSet {{   (127 223 )   }} scg.protocol.ExistsForAll {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (127
135 )   }} scg.protocol.ExistsForAll {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (35 127 )   }}
scg.protocol.ForAllExists {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (127 250 )   }} scg.protocol.ExistsForAll {{
}} 1.0 1.0  oppose  protocolResponse
teamName  {{ PRX }}
reputation 100.0

 request propose csp.CSPInstanceSet {{   (127 219 )   }} scg.protocol.ForAllExists {{      }} 1.0 1.0
csp.CSPInstanceSet {{   (127 223 )   }} scg.protocol.ExistsForAll {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (127
135 )   }} scg.protocol.ExistsForAll {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (35 127 )   }}
scg.protocol.ForAllExists {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (127 250 )   }} scg.protocol.ExistsForAll {{
}} 1.0 1.0  oppose csp.CSPInstanceSet {{   (127 219 )   }} scg.protocol.ForAllExists {{      }} 1.0 1.0
csp.CSPInstanceSet {{   (127 223 )   }} scg.protocol.ExistsForAll {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (127
135 )   }} scg.protocol.ExistsForAll {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (35 127 )   }}
scg.protocol.ForAllExists {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (127 250 )   }} scg.protocol.ExistsForAll {{
}} 1.0 1.0  protocolRequest
reply propose csp.CSPInstanceSet {{   (58 127 )   }} scg.protocol.NegativeSecret {{      }} 0.99 0.8999
csp.CSPInstanceSet {{   (72 127 )   }} scg.protocol.NegativeSecret {{      }} 0.99 0.8999  oppose refuting
refuting refuting refuting refuting  protocolResponse
teamName  {{ AnushRashmi }}
reputation 100.0

 request propose csp.CSPInstanceSet {{   (127 219 )   }} scg.protocol.ForAllExists {{      }} 1.0 1.0
csp.CSPInstanceSet {{   (127 223 )   }} scg.protocol.ExistsForAll {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (127
135 )   }} scg.protocol.ExistsForAll {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (35 127 )   }}
scg.protocol.ForAllExists {{      }} 1.0 1.0 csp.CSPInstanceSet {{   (127 250 )   }} scg.protocol.ExistsForAll {{
}} 1.0 1.0 csp.CSPInstanceSet {{   (58 127 )   }} scg.protocol.NegativeSecret {{      }} 0.99 0.8999
csp.CSPInstanceSet {{   (72 127 )   }} scg.protocol.NegativeSecret {{      }} 0.99 0.8999  oppose
csp.CSPInstanceSet {{   (58 127 )   }} scg.protocol.NegativeSecret {{      }} 0.99 0.8999 csp.CSPInstanceSet {{
(72 127 )   }} scg.protocol.NegativeSecret {{      }} 0.99 0.8999  protocolRequest provide csp.CSPInstanceSet {{
(127 223 )   }} scg.protocol.ExistsForAll {{      }} 1.0 1.0 provide csp.CSPInstanceSet {{   (127 135 )   }}
```

Figure H: Example of a raw history file

The raw history represents an unprocessed output of all the tournament transactions. Each block represents a turn for a particular avatar. These blocks get larger as the rounds progress because there is more work to be done in the later stages of a tournament. Therefore, it may be difficult to analyze the data contained in these files. However, these files are beneficial in that they are exhaustive (i.e., all the information regarding tournament transactions are recorded here verbatim).

7.2 Smart History


agentx.ccs.neu.edu:7007/...
agentx.ccs.neu.edu:7007/tournament_details?resource=smart_history&session=8B85DCEC416226A1&tournament

```
claim csp.CSPInstanceSet {{   (127 219 )   }} scg.protocol.ForAllExists {{      }} 1.0 1.0
proposer  {{ AnushRashmi }}
opposer  {{ PRX }}
action refuting
responses provider  {{ AnushRashmi }}  pr provide csp.CSPInstance {{  v1 v2 v3 v4 v5 v6 v7 v8 v9 v10  (219 {1}
v10 v9 v8 ) (219 {1} v10 v9 v7 ) (219 {1} v10 v9 v6 ) (219 {1} v10 v9 v5 ) (219 {1} v10 v9 v4 ) (219 {1} v10 v9
v3 ) (219 {1} v10 v9 v2 ) (219 {1} v10 v9 v1 ) (219 {1} v10 v8 v7 ) (219 {1} v10 v8 v6 ) (219 {1} v10 v8 v5 )
(219 {1} v10 v8 v4 ) (219 {1} v10 v8 v3 ) (219 {1} v10 v8 v2 ) (219 {1} v10 v8 v1 ) (219 {1} v10 v7 v6 ) (219 {1}
v10 v7 v5 ) (219 {1} v10 v7 v4 ) (219 {1} v10 v7 v3 ) (219 {1} v10 v7 v2 ) (219 {1} v10 v7 v1 ) (219 {1} v10 v6
v5 ) (219 {1} v10 v6 v4 ) (219 {1} v10 v6 v3 ) (219 {1} v10 v6 v2 ) (219 {1} v10 v6 v1 ) (219 {1} v10 v5 v4 )
(219 {1} v10 v5 v3 ) (219 {1} v10 v5 v2 ) (219 {1} v10 v5 v1 ) (219 {1} v10 v4 v3 ) (219 {1} v10 v4 v2 ) (219 {1}
v10 v4 v1 ) (219 {1} v10 v3 v2 ) (219 {1} v10 v3 v1 ) (219 {1} v10 v2 v1 ) (219 {1} v9 v8 v7 ) (219 {1} v9 v8 v6
) (219 {1} v9 v8 v5 ) (219 {1} v9 v8 v4 ) (219 {1} v9 v8 v3 ) (219 {1} v9 v8 v2 ) (219 {1} v9 v8 v1 ) (219 {1} v9
v7 v6 ) (219 {1} v9 v7 v5 ) (219 {1} v9 v7 v4 ) (219 {1} v9 v7 v3 ) (219 {1} v9 v7 v2 ) (219 {1} v9 v7 v1 ) (219
{1} v9 v6 v5 ) (219 {1} v9 v6 v4 ) (219 {1} v9 v6 v3 ) (219 {1} v9 v6 v2 ) (219 {1} v9 v6 v1 ) (219 {1} v9 v5 v4
) (219 {1} v9 v5 v3 ) (219 {1} v9 v5 v2 ) (219 {1} v9 v5 v1 ) (219 {1} v9 v4 v3 ) (219 {1} v9 v4 v2 ) (219 {1} v9
v4 v1 ) (219 {1} v9 v3 v2 ) (219 {1} v9 v3 v1 ) (219 {1} v9 v2 v1 ) (219 {1} v8 v7 v6 ) (219 {1} v8 v7 v5 ) (219
{1} v8 v7 v4 ) (219 {1} v8 v7 v3 ) (219 {1} v8 v7 v2 ) (219 {1} v8 v7 v1 ) (219 {1} v8 v6 v5 ) (219 {1} v8 v6 v4
) (219 {1} v8 v6 v3 ) (219 {1} v8 v6 v2 ) (219 {1} v8 v6 v1 ) (219 {1} v8 v5 v4 ) (219 {1} v8 v5 v3 ) (219 {1} v8
v5 v2 ) (219 {1} v8 v5 v1 ) (219 {1} v8 v4 v3 ) (219 {1} v8 v4 v2 ) (219 {1} v8 v4 v1 ) (219 {1} v8 v3 v2 ) (219
{1} v8 v2 v1 ) (219 {1} v7 v6 v5 ) (219 {1} v7 v6 v4 ) (219 {1} v7 v6 v3 ) (219 {1} v7 v6 v2 ) (219 {1} v7 v6 v1
) (219 {1} v7 v5 v4 ) (219 {1} v7 v5 v3 ) (219 {1} v7 v5 v2 ) (219 {1} v7 v5 v1 ) (219 {1} v7 v4 v3 ) (219 {1} v7
v4 v2 ) (219 {1} v7 v4 v1 ) (219 {1} v7 v3 v2 ) (219 {1} v7 v3 v1 ) (219 {1} v7 v2 v1 ) (219 {1} v6 v5 v4 ) (219
{1} v6 v5 v3 ) (219 {1} v6 v5 v2 ) (219 {1} v6 v5 v1 ) (219 {1} v6 v4 v3 ) (219 {1} v6 v4 v2 ) (219 {1} v6 v4 v1
) (219 {1} v6 v3 v2 ) (219 {1} v6 v3 v1 ) (219 {1} v6 v2 v1 ) (219 {1} v5 v4 v3 ) (219 {1} v5 v4 v2 ) (219 {1} v5
v4 v1 ) (219 {1} v5 v3 v2 ) (219 {1} v5 v3 v1 ) (219 {1} v5 v2 v1 ) (219 {1} v4 v3 v2 ) (219 {1} v4 v3 v1 ) (219
{1} v4 v2 v1 ) (219 {1} v3 v2 v1 ) (127 {1} v10 v9 v8 ) (127 {1} v10 v9 v7 ) (127 {1} v10 v9 v6 ) (127 {1} v10 v9
v5 ) (127 {1} v10 v9 v4 ) (127 {1} v10 v9 v3 ) (127 {1} v10 v9 v2 ) (127 {1} v10 v9 v1 ) (127 {1} v10 v8 v7 )
(127 {1} v10 v8 v6 ) (127 {1} v10 v8 v5 ) (127 {1} v10 v8 v4 ) (127 {1} v10 v8 v3 ) (127 {1} v10 v8 v2 ) (127 {1}
v10 v8 v1 ) (127 {1} v10 v7 v6 ) (127 {1} v10 v7 v5 ) (127 {1} v10 v7 v4 ) (127 {1} v10 v7 v3 ) (127 {1} v10 v7
v2 ) (127 {1} v10 v7 v1 ) (127 {1} v10 v6 v5 ) (127 {1} v10 v6 v4 ) (127 {1} v10 v6 v3 ) (127 {1} v10 v6 v2 )
(127 {1} v10 v6 v1 ) (127 {1} v10 v5 v4 ) (127 {1} v10 v5 v3 ) (127 {1} v10 v5 v2 ) (127 {1} v10 v5 v1 ) (127 {1}
v10 v4 v3 ) (127 {1} v10 v4 v2 ) (127 {1} v10 v4 v1 ) (127 {1} v10 v3 v2 ) (127 {1} v10 v3 v1 ) (127 {1} v10 v2
v1 ) (127 {1} v9 v8 v7 ) (127 {1} v9 v8 v6 ) (127 {1} v9 v8 v5 ) (127 {1} v9 v8 v4 ) (127 {1} v9 v8 v3 ) (127 {1}
v9 v8 v2 ) (127 {1} v9 v8 v1 ) (127 {1} v9 v7 v6 ) (127 {1} v9 v7 v5 ) (127 {1} v9 v7 v4 ) (127 {1} v9 v7 v3 )
(127 {1} v9 v7 v2 ) (127 {1} v9 v7 v1 ) (127 {1} v9 v6 v5 ) (127 {1} v9 v6 v4 ) (127 {1} v9 v6 v3 ) (127 {1} v9
v6 v2 ) (127 {1} v9 v6 v1 ) (127 {1} v9 v5 v4 ) (127 {1} v9 v5 v3 ) (127 {1} v9 v5 v2 ) (127 {1} v9 v5 v1 ) (127
{1} v9 v4 v3 ) (127 {1} v9 v4 v2 ) (127 {1} v9 v4 v1 ) (127 {1} v9 v3 v2 ) (127 {1} v9 v3 v1 ) (127 {1} v9 v2 v1
) (127 {1} v8 v7 v6 ) (127 {1} v8 v7 v5 ) (127 {1} v8 v7 v4 ) (127 {1} v8 v7 v3 ) (127 {1} v8 v7 v2 ) (127 {1} v8
v7 v1 ) (127 {1} v8 v6 v5 ) (127 {1} v8 v6 v4 ) (127 {1} v8 v6 v3 ) (127 {1} v8 v6 v2 ) (127 {1} v8 v6 v1 ) (127
{1} v8 v5 v4 ) (127 {1} v8 v5 v3 ) (127 {1} v8 v5 v2 ) (127 {1} v8 v5 v1 ) (127 {1} v8 v4 v3 ) (127 {1} v8 v4 v2
) (127 {1} v8 v4 v1 ) (127 {1} v8 v3 v2 ) (127 {1} v8 v3 v1 ) (127 {1} v8 v2 v1 ) (127 {1} v7 v6 v5 ) (127 {1} v7
v6 v4 ) (127 {1} v7 v6 v3 ) (127 {1} v7 v6 v2 ) (127 {1} v7 v6 v1 ) (127 {1} v7 v5 v4 ) (127 {1} v7 v5 v3 ) (127
{1} v7 v5 v2 ) (127 {1} v7 v5 v1 ) (127 {1} v7 v4 v3 ) (127 {1} v7 v4 v2 ) (127 {1} v7 v4 v1 ) (127 {1} v7 v3 v2
) (127 {1} v7 v3 v1 ) (127 {1} v7 v2 v1 ) (127 {1} v6 v5 v4 ) (127 {1} v6 v5 v3 ) (127 {1} v6 v5 v2 ) (127 {1} v6
v5 v1 ) (127 {1} v6 v4 v3 ) (127 {1} v6 v4 v2 ) (127 {1} v6 v4 v1 ) (127 {1} v6 v3 v2 ) (127 {1} v6 v3 v1 ) (127
{1} v6 v2 v1 ) (127 {1} v5 v4 v3 ) (127 {1} v5 v4 v2 ) (127 {1} v5 v4 v1 ) (127 {1} v5 v3 v2 ) (127 {1} v5 v3 v1
) (127 {1} v5 v2 v1 ) (127 {1} v4 v3 v2 ) (127 {1} v4 v3 v1 ) (127 {1} v4 v2 v1 ) (127 {1} v3 v2 v1 )   }}
 provider  {{ PRX }}  pr solve csp.CSPSolution {{  [ (v1 -> true) (v10 -> true) (v2 -> true) (v3 -> true) (v4 ->
true) (v5 -> true) (v6 -> true) (v7 -> true) (v8 -> true) (v9 -> true)  ]  }}
  winner  {{ PRX }}
pointsWon 1.0
```

Figure I: Example of a smart history file

The smart history represents a processed output of the tournament transactions. Each block represents a particular claim. A block contains chronological information that describes every stage of a claim from its initial proposal to the determination of a winner. Therefore, these blocks are relatively similar in size (unless some error has occurred between stages). Since this information is organized in a more reasonable manner, it is much easier to analyze than an equivalent raw history file. However, it may be less accurate as it is not a direct output of the game transactions.

8 Unresolved Issues

· Arguably the biggest unresolved issue is the inability to truly download a baby avatar file from the link on a tournament status page (see Section 6.4). Due to reuse of code from the legacy system, it was not discover until late in the design process that it was not possible to allow dynamic generation of the appropriate files for this feature. This would need to be rectified in future implementations as it is very non-user friendly to obtain a baby avatar using the current system.

· There was discussion earlier in the semester of having a system that would allow anybody to launch test tournaments (possibly to evaluate improvements to the logic of a sophisticated avatar or for some other purpose). As it currently stands, it is only possible for someone with the administrator credentials to sign into the web interface and create a new tournament. Therefore, it may be beneficial to allow regular users to create their own tournament in later revisions.

· Having an HTML based smart history would be a nice upgrade from the current implementation. There was some of this in the legacy system, but it was not carried over to the new implementation (possibly due to the fact that there were already some issues and bugs found in the smart history very late in the development process). It would be helpful for the smart history to be even easier to analyze than it already is.

· Creating a generic baby avatar was an idea that was discussed at some point but was never seriously followed up on. At the very least, this idea should probably be revisited in future classes to determine what the benefit of such a change might be. Additionally, there was a "`takeTurn`" method as part of an abstract Avatar class that would handle all the required actions inclusively. This was later changed back to using an interface approach, but there was no explanation why. If the generic baby avatar approach is not used, then changing back to an abstract class might be a solid alternative.

· This list of unresolved issues is itself somewhat unresolved. The last 4 points were from an individual's perspective of things that should be fixed or improved. However, the success of the system depends on more of a collective perspective. Members of the class should add to this section over the next few days so that everything is covered in detail and that future development can be streamlined to a long list of specific problems.