Specker Challenge Game: Requirements & Design From Generic to CNF/CSP Versions

Karl Lieberherr and Ahmed Abdelmeged and Bryan Chadwick

College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA.
{lieber,mohsen,chadwick}@ccs.neu.edu

Date: September 7, 2009, Fall 2009 version

1 Mathematical Contests in the Renaissance

Since 2006 we have been working on a game, called the Specker Challenge Game (SCG), both for our teaching and to push our research in programming tools.

Consider yourself moved back a few hundred years to the time of the Renaissance in Italy. The mathematicians challenged each other regarding the best algorithms to solve problems. Here is a story about who was the best in solving cubic equations in the 16th century. [1]

1.1 Contest about solving cubic equations

> The first person known to have solved cubic equations algebraically was del Ferro but he told nobody of his achievement. On his deathbed, however, del Ferro passed on the secret to his (rather poor) student Fior. For mathematicians of this time there was more than one type of cubic equation and Fior had only been shown by del Ferro how to solve one type, namely 'unknowns and cubes equal to numbers' or (in modern notation) $x^3 + a * x = b$. Fior began to boast that he was able to solve cubics and a challenge between him and Tartaglia was arranged in 1535. In fact Tartaglia had also discovered how to solve one type of cubic equation since his friend Zuanne da Coi had set two problems which had led Tartaglia to a general solution of a different type from that which Fior could solve, namely 'squares and cubes equal to numbers' or (in modern notation) $x^3 + a * x^2 = b$. For the contest between Tartaglia and Fior, each man was to submit thirty questions for the other to solve. Fior was supremely confident that his ability to solve cubics would be enough to defeat Tartaglia but Tartaglia submitted a variety of different questions, exposing Fior as an, at best, mediocre mathematician. Fior, on the other hand, offered Tartaglia thirty opportunities to solve the 'unknowns and cubes' problem since he believed that he would be unable to solve this type, as in fact had been the case when the contest was set up. However, in the early hours of 13 February 1535, inspiration came to Tartaglia and he discovered the method to solve 'squares and cubes equal to numbers'. Tartaglia was then able to solve all thirty of Fior's problems in less than two hours. As Fior had made little headway with Tartaglia's questions, it was obvious to all who was the winner. Tartaglia didn't take his prize for winning from Fior, however, the honour of winning was enough.
>
> At this point Cardan enters the story. As public lecturer of mathematics at the Piatti Foundation in Milan, he was aware of the problem of solving cubic equations, but, until the contest, he had taken Pacioli at his word and assumed that, as Pacioli stated in the Suma published in 1494, solutions were impossible. Cardan was greatly intrigued when Zuanne da Coi told him about the contest and he immediately set to work trying to discover Tartaglia's method for himself, but was unsuccessful. A few years later, in 1539, he contacted Tartaglia, through an intermediary, requesting that the method

[1] Thanks to Shriram Krishnamurthi from Brown University for pointing this out. The following text is from http://www.gap-system.org/ history/Biographies/Tartaglia.html

could be included in a book he was publishing that year. Tartaglia declined this opportunity, stating his intention to publish his formula in a book of his own that he was going to write at a later date. Cardan, accepting this, then asked to be shown the method, promising to keep it secret. Tartaglia, however, refused.

And the story goes on about how Cardan tricked Tartaglia to reveal his secrets. The mathematical contests had a good effect on scientific progress. We can get a similar effect on scientific progress for computational processes today. But we need to design contest rules appropriate for the 21. century.

2 Computational Contests in the 21. Century

The key idea is still the same: we want to determine who is the best at solving a computational problem. What is different now is that we have fast machines to help us solve the problems. So in the 21. century we compete by putting our computational knowledge into software, called agents, and we let the agents compete.

Like in the Renaissance, the agents pose hard problems to each other; problems that the other is unlikely to solve. This is important to winning a contest: it is not just being good at solving problems but also posing hard problems. This is an important idea to improve the current benchmark approach that is widely used. In the benchmark approach, a fixed set of problems is given to all contestants.

Like in the Renaissance, the agents will exploit each other's weaknesses and this is a great source of learning and progress for the programmer teams behind the agents.

Like in the Renaissance, the agents will try to hide the secrets of their agents to use them in future competitions. To bring more progress to the game, we will force the agents to reveal their secrets after a fixed number of contests. So in the 21. century, Tartaglia could keep his secret only for a few weeks.

Like in the Renaissance, an administrator controls the game. Both Fior and Tartaglia had to submit 30 problems of the agreed upon kind to the administrator. The administrator made sure that the rules of the contest are followed. The adminstrator in the agent contest is of great importance because it ensures that the rules of the contest are followed religiously by all agents. If an agent misbehaves, it is thrown out from the game.

Like in the Renaissance, the contestants agree on a computational problem that is the subject of the contest. The agents must propose problems in this domain.

Unlike in the Renaissance, we can have now daily contests between the agents. This will throughly test the agents and lead to faster progress. The only limiting factor is how fast the programmer teams can improve their agents.

Unlike in the Renaissance, agents will pose not individual problems to each other but entire families of problems, defined by a predicate, will be posed to other agents. This makes the game more interesting and requires a better understanding of the underlying computational process.

Tartaglia might pose the following challenge to Fior: I will give you a problem x of kind X (but I don't tell you now which one). If you can solve problem x with quality q, you win. Fior can refuse the problem. But if he accepts it, Tartaglia has to give him a hard instance where Fior cannot achieve quality q. This kind of challenge is posed in the classic Specker Challenge Game.

Tartaglia might pose the following challenge to Fior: I will give you a problem x of kind X (but I don't tell you now which one). If you accept this challenge, I will give you $x \in X$ and also the quality $q(x)$ of the solution that I have for problem x. If you can solve x with quality $q(x) * p$ for some constant p, you win, otherwise you lose. For X, p is a constant less than or equal to 1. Fior can refuse the problem. But if he accepts it, Tartaglia has to give him a hard instance where Fior cannot achieve quality $q(x) * p$. This kind of challenge is posed in the secret Specker Challenge Game. It is called secret Specker Challenge Game, because Tartaglia knows a secret solution that Fior has to approximate sufficiently well.

The game rules are that in each round, each player poses at least one challenge and accepts at least one challenge or she proves that all challenges are impossible to solve by reoffering them under more favorable conditions.

2.1 Specker Challenge Game Contest Design

The contest design operates at 3 levels of abstraction. At the highest level is the generic administrator that enforces the rules of an artificial world inhabited by agents. The agents must regularly offer challenges and accept challenges. The generic administrator keeps track of the scores of each agent and can be used with many different game instances.

To test the generic administrator, a generic baby agent is developed. The baby agent has the basic capability to follow all game rules and survive at least for a short period in the artificial world. When the generic administrator and a few generic baby agents are instantiated by the same game instance, we get a very simple game to test the administrator and baby agents. The baby agents are randomized; therefore they all will behave differently although they execute the same code.

The second level is the design of a specific game instance. The motivation for designing a specific game may come from many groups:

- Researchers
  who want to objectively measure and compare the quality of their understanding of a computational problem. The Specker Challenge Game complements the static benchmark technology. An editor of a special issue of a competitive journal might design a game to determine the current state of the art in a specific computational domain.
- Software Developers
  who want to thoroughly test and evaluate their software. The SCG will test the software through the rules of the game: agents can score winning points when they exploit a weakness in another agent.
- Educators
  who want to use an experiential teaching approach where the students write self-sufficient agents that compete with each other. Students learn about software development as well as the computational problem being solved in the game instance. The amount of software that needs to be developed is significantly reduced by giving the students a baby agent that can already walk and talk. Then the students need only to add "intelligence" to the baby agent.
  The amount of software that needs to be produced can be further reduced by offering a component market of software components that are useful for the given computational problem. The component market could also be student developed: agent teams that develop successful components would receive additional points in addition to the points they win in the contests.
  The benefits of SCG for education are: (1) it is fun: developing an agent that can survive on its own in a "real" artificial world inhabited by agents developed by peers is rewarding; (2) students are competitive and this competitiveness fires their learning process: they want their agent to win; and, of course, they have to be better than the baby agent. (3) students help students: because the game rules encourage exploiting weaknesses in other agents, the students get plenty of feedback about their agent from each contest. Indeed, each (full round-robin) contest provides an objective grade for each student. (4) it is an ideal topic for a CS capstone course: students are integrating knowledge from different areas of CS: Software Design / Programming, Discrete Structures, Theory of Computation, Calculus, etc.
- Groups of Educators
  who want to objectively evaluate their students and compare students from different universities regarding their knowledge of a computational problem and their skills to translate computational knowledge into a self-sufficient software agent. The faculty members accept the challenge of researching the computational problem and passing their insights on to their students so that they will put the winning intelligence into their agent.

A game instance is informally defined by a definition of a computational process (input/output definitions), and a description of the challenges that may be used in the game. Those descriptions are fed to a generic Specker Challenge Game administrator. The result is a specific administrator that is ready to accept agent contestants.

The third level is the agent level where small teams of computer scientists pour all their knowledge about the chosen computational process and the challenges that may be posed, into their agent.

2.2 Communication Design

The agents and the administrator need to communicate. There are many ways to implement this communication: e.g., using a file system that acts as a blackboard or using the web.

If the web is used, the administrator is the web agent and the agents turn into web applications that respond to the administrator. When it is time for a competition, the administrator broadcasts the computational problem for which a competition will be held. The web applications that are ready for the contest with the given game instance register with the administrator and the fun begins.

The web version of the Specker Challenge Game has the important advantage: the contests become automated and it becomes easy to run many of them. The partial results of a competition are posted on the web during the contest.

2.3 SCG Contest Services Inc. (SCG CSI)

This role is currently played by the Demeter Research Group at Northeastern University that does research in software development. SCG CSI develops and maintains the generic administrator and generic baby agent and their documentation. SCG CSI publishes several game instances that have been proven useful in pushing either the state of the art in research or teaching.

SCG CSI offers game instance design services to help companies design a suitable game that brings forth the best computational process that they need for a new application.

SCG CSI offers game instance design services to help educators design a suitable game that covers the educational objectives that need to be covered.

SCG CSI offers contest execution services where SCG CSI monitors the contests for proper adherence to the chosen security policy. SCG CSI will deal with situations where agents try to attack the administrator to cheat.

SCG CSI offers contest data mining tools that are useful to agent developer teams. Finding weaknesses in other agents depends on studying their behavior through analysis of the game histories.

SCG CSI offers support for stable interfaces so that "old" agents may compete in contests for the same game instance 10 years from now.

For the Specker Challenge Game game we currently use the broad domain of combinatorial optimization problems. We will select specific families of combinatorial optimization problems.

3 The Specker Challenge Game (SCG)

The Specker Challenge Game is a game of interacting agents that seek to maximize their energy by engaging in energy enhancing interactions. A potential interaction is described by a description of raw materials that may be delivered and a price. When an interaction is entered, the price is paid (energy deducted), raw materials are delivered and finished and the quality achieved for the finished product is paid (energy added).

The Specker Challenge Game is interesting because it provides a very simple model of the behavior of living organisms who choose actions that give them positive energy in the long run. Those organisms need to look ahead of what the consequences of their actions might be. When they perform an action in the world, for example, to offer or buy a service, they need to judge whether this action might drain their energy.

We use the following terms as synonyms: challenge / service / derivative; life energy / money; price / cost of accepting challenge; raw material / instance of computational problem; finished product / raw material with solution.

This document describes bothe the Classic and Secret Specker Challenge Game game. More information on the Secret Specker Challenge Game is here: http://www.ccs.neu.edu/home/lieber/evergreen/specker/sdg-home.html.

4 Definition

We parameterize the *Specker Challenge Game* over the challenges, raw materials etc. to be used. Then we discuss the generalized satisfiability instance (CSP). This supports better separation of concerns at the requirements level. We can then formulate the important game rules without knowing the details of the challenges.

4.1 Parameterization

The Specker Challenge Game game is parameterized by a tuple $C = (G, Pred, D, R(d, d \in D), O(r, r \in R(d), d \in D), F, Q)$, where G is a set of raw materials, *Pred* is a set of *predicates*, each selecting a subset of G, D (*challenges*) is a set of pairs $d = (t, p)$, where $t \in Pred$ is the predicate $pred(d)$ and $0 \leq p \leq 1$ is the price $price(d)$. $R(d)$ (*raw materials* for challenge d) is a set with each element $r \in R(d)$ satisfying predicate $pred(d)$, the predicate of challenge d. $O(d)$ is the set of feasible solutions for elements in $R(d)$; we denote a feasible solution for raw material r as $o(r) \in O(d)$. F (*finished products*) is a set of pairs $(r, o(r))$, with $r \in R$ and $o(r) \in O$. Q (*quality*) is a function that maps a finished product $(r, o(r))$ to $[0, 1]$. *Pred* is expressed in a suitable chosen predicate language described by a grammar and its semantics.

The game is about buying and selling challenges. When a challenge $d = (t, p)$ is offered, only its predicate t and price p are known. The creator and buyer of a challenge need to do a *min-max* analysis for the predicate t, which makes the game interesting. By min-max analysis we mean that it must be known what the worst-case raw material is for a given predicate. The seller must know this to price the challenge properly and the buyer must know this to decide whether the price is right. The buyer of a challenge will be paid the *quality* of the finished product achieved for the raw material produced by the seller, after the challenge was bought.

SpeckerChallengeGame (C) is a tuple $(P, account, store, config)$, where P is an ordered set of players, *account* is a function that assigns a positive real number to each player. *store* is a function that assigns a pair $(forSale, bought)$ to each player, where *forSale* is a set of challenges for sale and *bought* is a set of tuples $(d, seller, r, f)$, where d is a challenge, $seller \in P$, $r \in R \cup \{absent\}$ and $f \in F \cup \{absent\}$. $config$ is a tuple $(init, maxTurns, timeslot, mindec)$, which configures the game. This configuration tuple will be later expanded with specifics for the combinatorial structure[2] to be used in the game. *init* is a positive real number, giving the initial amount of the account of each player in P. *maxTurns* is the maximum number of turns the game will be played. *timeslot* is the amount of time given to each player. *mindec* is a small real number which specifies the minimum decrement when the price of a challenge is lowered.

4.2 Two Player Game Rules: explained with Chess analogy

Note: The two player game rules are listed here to explain the game using a chess analogy, but the game that is implemented uses a different protocol. The game that is implemented also works with only two players.

Without loss of generality, we limit the game to two players, called White and Black, as in chess. White starts the game. The board is a set G of raw materials and a predicate language to express predicates selecting subsets of raw materials.

A mega move White/Black consists of: White's first move is to offer a set of challenges for sale. Black's next move is to decide which subset of the challenges on sale to buy. White's next move is to deliver raw material for the bought challenges. Black's next move is to finish the raw materials that have been delivered. If Black does not buy any challenges, there is no raw material delivery and no finished products delivered but instead Black reoffers all challenges for sale by White at a lower price.

After a mega move White/Black comes a mega move Black/White with the roles reversed. An even number of mega moves will be played, determined by the configuration object. The winner of the game is the agent with the most life energy. If both have the same life energy, it is a tie.

The game rules of the next subsection apply, adapted to the the two player situation.

[2] CNF or CSP for example.

4.3 Game Rules

The *SpeckerChallengeGame* (C) game has an asynchronous and a synchronous version. Here we define the synchronous version where players operate sequentially; in the asynchronous version, players can buy and sell at any time.

The rules of the synchronous *SpeckerChallengeGame* (C) game are:

1. *Main Objective.* The winner of the game is the player with the most money in the player's account at the end of the game. The account value ranks the players. Players with the same account value have the same rank.
2. *Uniform Turns.* Only one player is playing at a given time. When a player is done, the next player is the next element in the ordered set P. A player may indicate that s/he is done in a variety of ways, for example by creating a file. In other words, the players take turns in a uniform sequence.
3. *Accept or Re-offer.* To make challenges more attractive, on each turn, a player must accept at least one challenge offered for sale by other players or re-offer all currently offered challenges by all other players at a lower price. When a challenge is accepted the seller is paid by the buyer the price of the challenge.
4. *Offer new challenge.* On each turn, a player must offer a challenge whose predicate does not exist yet in the store of offered challenges, or whose price is lower than the price of all other challenges of the same predicate in the store.
5. *Timely delivery.* A player must deliver a problem, called a raw material problem, for challenges bought from them in the previous round. The raw material problem must match the predicate of the challenge.
6. *Obligation to the finished product.* The owner of a challenge is obliged to pay for a finished product, i.e., the solution to the raw material problem, based on the solution quality as soon as the solution is delivered.
7. *Price lowering.* When lowering the price of a challenge, it must be lowered at least by *mindec*.
8. *Bought once.* A challenge may only be bought once from the store.
9. *Positive account.* Players must maintain a positive account.
10. *Time limit.* Players must finish within *timeslot*.
11. *Consequences.* Players that don't comply with the rules are removed from the game. If a player is removed from the game, its challenges are removed from the store and its bought, but unfinished challenges are refunded to the buyer.
12. *Completion.* After *maxTurns* rounds, nothing can be bought but raw material problems are still delivered and solved until all outstanding obligations are met.

4.4 Archiving Concern

We want to be able to archive games for further analysis. We use the following 4 archiving transactions.

1. When a player p offers a challenge d, we archive $create(p, d)$.
2. When a player p buys a challenge d from $seller$, we archive $buy(seller, p, d)$.
3. When a player p delivers raw material r for challenge d, we archive $deliverR(p, d, r)$.
4. When a player p delivers the finished product $f = (r, o(r))$ for raw material r and challenge d, we archive $deliverF(p, d, f)$.

Given a sequence of archived transactions, we can check whether the players followed the rules of the game. For example, a player can only finish raw material for a challenge that was bought previously.

4.5 Security Concern

It is important to have integrity of the competitions. We rely on the administrator to enforce the rules of the artificial world. If it fails to provide a safe and equal playground for all agents, the intelligence of the agents is of questionable value and we start mistrusting the game and feel cheated.

A agent, to gain life energy quickly, may try the following:

– Attack the administrator.
  There may be vulnerabilities in the administrator that can be used to gain life energy. An example would be to directly manipulate the account values in one's favor or to manipulate the store.
  These attacks are undesirable unless one wants to strengthen the administrator.
– Attack another agent.
  There may be vulnerabilities in a agent that can be exploited to gain life energy. We distinguish between intelligence and mechanism attacks. An example of an intelligence attack is to trade challenges for which the attacked agent makes poor decisions. An example of a mechanism attack is to modify a "personal" file of the attacked agent.
  - Intelligence
    This kind of attack is desirable because it improves the quality of the agents. The attacking agent will gain life energy for alerting the attacked agent of its vulnerabilities. But in the long run it will improve the quality of all agents.
  - Mechanism
    In a course where software security is taught, this is also a dersirable kind of attack because it improves the security of the agents.

A good approach to security is to use different security policies for different competitions. And to allow agent submissions for different policies. Although the administrator in principle should be perfect and not contain vulnerabilities, there might be some hidden threats.

There are three kinds of attack modes: Administrator Attack Mode, Agent Attack Mode (Intelligence) and Agent Attack Mode (Mechanism). Each of the attack modes may be on or off is described by a Boolean triple.

Security policy: When an attack mode is off, there is a gentleman's agreement that no attack of that kind takes place. If it does, there is a penalty for the offending agent, if caught. The history files of all competitions are public which allows other agents to look for violators. The resolution of an issue might require an arbitration hearing where the software developers of a agent act as its lawyers that defend the actions of the agent. A student acts as chair person of the arbitration panel and all class members that are not acting as lawyers act as members of the arbitration panel. In this sense, in class arbitration hearings are used as a security mechanism to resolve security disputes.

(0,0,0) is a very friendly agent that tries to win through uniform intelligence. The agent is not allowed to target individual agents.

(0,1,0) is a friendly debugging agent that exploits intelligence weaknesses of another agent. For example, the agent might always use the same challenges against the attacked agent because the attacked agent makes bad decisions regarding those challenges.

(0,0,1) is a friendly debugging agent that exploits mechanism weaknesses of another agent. For example, the attacking agent might modify a temporary file that the other agent left unprotected.

(1,0,0) is a vicious agent that tries to modify the artificial world in its favor.

(1,1,1) is a fierce agent.

Competitions often take place under the (0,1,0) policy. It makes sense to have (0,1,1) policies to make the agent developers aware of security problems in their agent. Competitions using the (1,*,*) security policy don't make sense if the objective is to improve the agents and not the administrator.

However, teams may submit two agents, one for the (0,1,0) or (0,1,1) or (0,0,1) policy and one for the (1,*,*) policy, illustrating an exploit in the administrator. Only the first will be used in the next competition while the second, if revealing a bug in the administrator, will be awarded $ 1 million in life energy.

The security concern in the game can only be partially addressed by the administrator through technical solutions. But a correct and secure administrator are an important part of the solution.

The game has the property that after each round, all information becomes public by exposing it in the history file. This reduces the need for cryptographic techniques.

5 Specialization

5.1 For CNF

This specialization initiates learning about propositional calculus and basic combinatorics. Algorithms for MAX-SAT are important to play the game well.

G is the set of all conjunctive normal forms that use only two clause types. $Pred = \{r1, r2\}$ where $r1$ and $r2$ are *clause types*. In other words, we express the predicates with clause types. A clause type is a pair (l, p), where p is a set of positive literals with l elements. $R(d)$ is the set of weighted CNF-formulas satisfying the predicate of d, where each clause has a positive integer *weight*. $O(r)$ is the set of all assignments for a CNF-formula = raw material r. F is a pair (r, J), where $J \in O$ is an assignment for the CNF-formula r. $Q(r, J)$ is the weighted fraction of satisfied clauses in CNF-formula r under assignment J.

To make the sets finite we add the following configuration tuple:

$$(maxVars, maxClauses, maxWeight, maxClauseLength)$$

5.2 For CSP

This specialization initiates learning about Boolean constraint satisfaction. Algorithms for MAX-CSP are important to play the game well.

For this formulation, G is the set of all Boolean weighted CSP-formulae that use only Boolean relations of arity 3. $Pred$ is the set of all subsets of the set of boolean relations of at most arity 3. There are 256 relations of arity 3. There are 2^{256} different predicates in $Pred$. $R(d)$ is the set of CSP($Pred$)-formulas that satisfy the predicate of d. A challenge is a set of relations in $Pred$ and a price. $O(r)$ is a Boolean assignment J for a CSP($Pred$)-formula r. F is a pair (r, J), where J is an assignment to the variables of r. $Q(r, J)$ is the weighted fraction of satisfied constraints in r under assignment J.

To make the sets finite we add the following configuration tuple:

$$(maxVars, maxConstraints, maxWeight)$$

5.3 Simpler Versions for Classic SCG CSP

We equate the Classic SCG CSP game to baseball which has two simpler versions: T Ball and Softball. Softball itself has two versions: slow pitch and fast pitch. Before our agents can play the full version of the Classic SCG CSP game, they need to learn simpler versions to ensure a smooth learning curve both for the agents and their creators.

Classic SCG CSP T Ball Playing T Ball is much easier than playing the full game. Classic SCG CSP T Ball has the same rules as Classic SCG, but for all challenges only one Boolean relation is allowed in the predicate.

Skills needed include: deriving expected fraction of satisfied constraints for a coin with bias b. Find the maximum bias. Generate all combinations of k elements out of n.

Classic SCG CSP Slow Pitch Softball Classic SCG CSP Slow Pitch Softball has the same rules as Classic SCG, but for all challenges the relations defining the predicate must form an implication tree. For example with the standard relational encoding used by

`http://www.ccs.neu.edu/home/lieber/courses/csu670/sp09/source/IR-2.0/doc/`

2 implies 6 implies 22 and 2 implies 10 is an implication tree with root 2. (Or for Classic SCG SAT: (1,1) implies (2,2) implies (3,3) is an implication tree.)

Slow Pitch Softball can be reduced to T Ball by recognizing that all but one relation in a fastpitch softball challenge are noise. Only the root of the tree is important.

An alternative definition of Slow Pitch Softball is that all but one relation can be eliminated using implications.

Classic SCG CSP Fast Pitch Softball This is the full version of Classic SCG CSP. We have multiple relations and we need to identify the important ones for determining the break-even price. There may be several implication relationships. For example, you might have a challenge d = (2,6,22,1) which consists of two implication chains: 2 implies 6 implies 22 and the singleton chain 1. An interesting question is whether to price (2,6,22,1) it is sufficient to price (2,1).

We define the simplified form: Level k Independent of Classic SCG CSP Fast Pitch Softball. In Level k Independent we have k relations so that no relation implies any of the others. An example of Level 2 Independent is (using the SAT notation) ((1,1) (2,0)) which has a break-even price of (sqrt(5)-1)/2 = 0.618 ...

Classic SCG CSP Fast Pitch Softball can be reduced to Classic SCG CSP Fast Pitch Softball Level k Independent (for some k) by finding the implications and eliminating the stronger relations.

Note that Classic SCG CSP Fast Pitch Softball Level 1 Independent is the same as T Ball.

We define the simplified form: Level k Reduced of Classic SCG CSP Fast Pitch Softball. We have any number of relations that can be reduced to Level k Independent.

An example: The Classic SCG CSP Fast Pitch Softball challenge (using SAT notation) ((1,1) (2,2) (3,3) (2,0) (3,0)) is Level 2 Reduced. It can be reduced to ((1,1) (2,0)) of Classic SCG CSP Fast Pitch Softball Level 2 Independent.

Therefore, ((1,1) (2,2) (3,3) (2,0) (3,0)) also has a break-even price of (sqrt(5)-1)/2 = 0.618 ...

Note that Classic SCG CSP Slow Pitch Softball is a special case of Classic SCG CSP Fast Pitch Softball Level 1 Reduced.

SCG Base Ball This game uses not only classic challenges but also secret challenges. It covers a much wider space than Classic SCG CSP because we can use in principle any NP-hard combinatorial maximization problem. One challenge is to define a generic language to define combinatorial maximization problems.

SCG CSP Base Ball is the special case where we stay with CSP but allow secret CSP challenges. This extension is relatively easy to add to Classic SCG CSP Fast Pitch Softball.

5.4 Version for Optimization

This generalizes the game to any standard optimization problem: an optimization problem is the problem of finding the best solution from all feasible solutions.

Note: This is very similar to the parameterization described in 2.1.

More formally, an optimization problem A is a quadruple (G, f, m), where

- G is a set of instances. These are our raw materials and in our world we will define them by a class dictionary and a semantic checker. An example would be a class dictionary for weighted CSP instances. The semantic checker would check, for example, that no variable appears more than once in a constraint.
- O is a function that maps an instance x of G to a set of feasible solutions. The set of feasible solutions for a weighted CSP instance with variable set V is the set of sets of $literals(V)$., which represents the set of all 2^n assignments for n variables.
- $m(x, y)$ is a function that maps an instance x and a feasible solution y of x to a real number. For a weighted CSP instance $m(x, y)$ is the weighted satisfaction ratio of assignment y for instance x.
- g is the goal function and is either min or max. For Max CSP, g is max.

A predicate on the raw materials is used to define a challenge. This is best done through a class dictionary and a semantic checker. For a weighted CSP raw material, the predicate might be that the instance contains only a given set of relations.

This abstract way of defining raw materials and predicates makes it much harder for agents to make their buying decisions. Making a buying decision involves analyzing the class dictionary for I and the code for f, m and g. It is good to specify the code in a "normalized" way, for example, using DemeterF.

To make the problem tractable, the optimization problem will be a constant for each competition. But different competitions will use different optimization problems. For example, we might change g from max to min.

When the optimization problem is a constant, it is the software developers who need to analyze the definition of the optimization problem and not the agents. The agents only need to be aware of a small number of optimization problems that are competition candidates.

Pseudo Boolean CSP We give another example for Optimization: We consider a version of CSP where the constraints are expressed as pseudo Boolean constraints.

The class dictionary is:

```
PseudoBooleanCSP =
  <variables> List(Variable) <constraints> List(PBConstraint).
...
Literal = <kind> Kind <v> Variable.
Kind = Pos | Neg.
Pos = .
Neg = "!".
Assignment = <literals> List(Literal).
```

We need a semantic checker to define the legal instances. The variables in part variables must be exactly the list of variables appearing in constraints.

All List(Literal) of the same length as variables is the set feasible solutions.

...

6 Put elsewhere

There is the issue that the web applications might not "understand" the optimization problem because there are many different ways to define the same optimization problem. We assume that there is a coordination between the administrator and web applications so that the web applications can check for equality between the optimization objects.

We need a class dictionary for class dictionaries (CDCD-SCG) and a class dictionary (DemeterF-SCG) for defining f, m, and g. Is CDCD-SCG equal to the cd for cds of DemeterF?

7 Reflection on the course

We can recognize elements of the game in real life. You have selected this course based on a a set of features: course description, college requirements, reputation, etc. Based on those features you have selected to take my course. The features correspond to the predicate of a challenge which you buy only by knowing the predicate. Now I deliver raw materials (subprojects) within the constraints of the course features we agreed upon. While we make the raw materials a bit hard for you, we do this with the objective of challenging and expanding your intellectual capabilities in software development. (This is different than in the game where we find hard raw materials to avoid a high payout.) You finish the raw materials by finding solutions to the subprojects and you will be paid by the quality of your finished product (the quality of your agent will strongly influence your grade). Your real payout, however, is the set of new skills you learn about software development.

You can learn about managing software development by learning the theory and by practicing the management of a software development project. It is my belief that without having experienced a project, you won't absorb the theory well. That is why we experience the project of developing algorithmic players.

8 Design

The requirements don't specify how the players communicate with each other. There are two options: a centralized versus a decentralized design. We use a centralized design using an administrator. The players and the administrator communicate through XML documents.

9 Implementation

We use Java as implementation language and a Java data binding tool to automatically generate parsers and printers from an XML schema.

In the implementation we practice Adaptive Programming (AP). In AP, as little structural information as possible is spread as narrowly as possible. (A good design principle in general not only for structural information.)

10 To Integrate

```
What happens if we use the model: challenge = (Pred, price, quality)
where quality is a function that says which quality we must reach to win.

Q:  quality of finished product found by buyer
SQ: quality of the secret finished product of creator
P:  price of challenge

quality may depend on SQ and price.
e.g., quality = SQ * price.

secret:
========================================
win = Q >= quality


profit (after paying price P) =

 if win then P*(1 + INTEREST) + INCENTIVE(Q - quality) else 0

 where INTEREST is a small percentage we get paid because
 we had to spend P for a round to get the challenge.
 We set INTEREST = 10% = 0.1.
 INCENTIVE is a factor that determines the reward for
 doing better than what we are supposed to do.

Problem: price not important?

Duc suggests:

profit = 2Q - SQ - P + 0.1

This formula will require a \robot to be smart in not
only buying and finishing the product.
It takes into consideration the price as it was intended
to be. So even if the buyer finishes at a higher
quality than the seller, they can still lose
money if they can't beat the price (which means they are stupid in buying).

This formula also provides incentives for \robots to actually
play the Secret Derivative variation.
As a buyer, they can earn even more money if their
```

```
finishing agent is better than the seller finishing agent.
As a seller, they can lower their loss on the buying back
of the finish product if they can finish much better than the buyer.
Heck, if the buyer is really dumb, they buyer will actually have to pay even more to the seller.
```

related work:

Okamoto, M., Kita, H., Ono, I., Kiga, D., Terano, T., Yamada, T. & Koyama, Y. (2008). Project-Based Learning of Computer Programming Using an Artificial Market System. In Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2008 (pp. 4545-4553). Chesapeake, VA: AACE.

U-MART project: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01419804

Acknowledgements: This research is supported by a grant from Novartis and and a grant from GMO.

Feng Zhou and Milena Georgieva Dimitrova have given detailed feedback on the requirements and they have written the previous version of the requirements.

11 Appendix: Implementation Notes

The implementation of administrator and baby agent of Spring 2009 is specific to Maximum Boolean CSP with the List(RelationNr) predicate to define challenges.

We want to eventually move from the Spring 2009 implementation to a generic administrator and a generic baby agent that are both parameterized by a maximization problem O.

With the general set-up, designing a baby agent becomes interesting. In the absence of any information about the parameters of the optimization problem, the following seems appropriate.

There are 4 subagents: offer, buy, createRM, finish. To implement those subagents requires the generation of random objects that satisfy a given property.

```
finish(i):
repeat
  select random feasible solution for i;
  compute quality;
until timeout

offer:
randomly choose predicate(p)
repeat
  randomly choose instance i satisfying predicate p;
  solve(i);
until timeout
offer challenge: (p, price)
where price is determined by loop.

accept challenge = (p, price):
similar to offer

createRM(p):
repeat
  randomly choose an instance i satisfying p;
  solve(i);
until timeout
return worst instance
```

```
language for defining object properties:

per field
  if List: length: NumberConstraint(Integer)
   randomly choose length
           element: List(ElementConstraint)
   (randonly choose element satisfying ElementConstraint)
  if abstract: (randomly choose alternative)
  if concrete: one choice only
     if Integer: NumberConstraint(Integer)
     if Float: NumberConstraint(Float)
     if Ident: regexp
     if String: regexp


NumberConstraint(T) :
  Single(T) | Range(T) | Enumeration(T) | Ref.
Single(T) = <v> T.
Range = "range" <low> T <high> T.
Enumeration(T) = "enum" <l> List(T).
Ref = <v> QualIdent.
QualIdent = <first> Ident <rest> List(DField).
DField = "." Field.

ElementConstraint : Distinct | In | Subset.
In = "in" Ref.
Subset = "subset" Ref.
Distinct = "distinct".

apply to CSP:

predicate: <rns> List(Integer)

CSP:
<vs> List(Variable)
<cs> List(Constraint).

Constraint = <w> Weight <rn> Integer <vars> List(Variable).
property: rn in predicate.rns
property: vars subset start.cs




refer to earlier fields in predicate or current cd.
```

remove: Problems we need to solve:

Given a cd, generate a random instance. Problem: cd has a semantic checker. We might need to generate too many random instances until we generate one that satisfies p.

New DGP function: Given a cd, generate a random object satisfying the cd.
==========
We attempt a full parmeterization gradually.