Specker Derivative Game Requirements & Design Generic to CNF/CSP Versions

Karl Lieberherr, Feng Zhou

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

1 Generic Requirements

We parameterize the *Specker Derivative Game* (SDG) over the derivatives, raw materials etc. to be used. This supports better separation of concerns at the requirements level. We can then formulate the important game rules without knowing the details of the derivatives.

1.1 Parameterization

The SDG game is parameterized by a tuple $C = (Typ, D, R, O, F, Q)$, where *Typ* is a set[1] of *types*, D (*derivatives*) is a set of pairs $d = (t, p)$, where $t \in Typ$ and $0 \leq p \leq 1$. R (*raw materials*) is a set with each element $r \in R$ having a type $typ(r) \in Typ$. O is the set of outcomes for elements in R; we denote an outcome for r as $o(r) \in O$. 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 $[0, 1]$.

The game is about buying and selling derivatives. When a derivative $d = (t, p)$ is offered, only its type t and price p are known. The creator and buyer of a derivative need to do a *min-max* analysis for the type 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 type. The seller must know this to price the derivative properly and the buyer must know this to decide whether the price is right. The buyer of a derivative will be paid the *quality* of the finished product achieved for the raw material produced by the seller, after the derivative was bought.

$SDG(C)$ 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 derivatives for sale and *bought* is a set of tuples $(d, seller, r, f)$, where d is a derivative, $seller \in P$, $r \in R \cup \{absent\}$ 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 derivative is lowered.

1.2 Game Rules

The $SDG(C)$ 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 $SDG(C)$ 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.

[1] All sets discussed are finite.
[2] CNF or CSP for example.

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. *Buy or Re-offer.* To make derivatives more attractive, on each turn, a player must buy at least one derivative offered for sale by other players or re-offer all derivatives for sale by all other players at a lower price. When a derivative is bought the seller is paid by the buyer the price of the derivative.
4. *Offer new derivative.* On each turn, a player must offer a derivative whose type does not exist yet in the store, or whose price is lower than the price of all other derivatives of the same type in the store.
5. *Timely delivery.* A player must deliver raw materials for derivatives bought from them in the previous round. The type of the raw material must match type of the derivative.
6. *Obligation to the finished product.* The owner of a derivative is obliged to pay for a finished product based on the quality achieved as soon as the finished product is delivered.
7. *Price lowering.* When lowering the price of a derivative, it must be lowered at least by *mindec*.
8. *Bought once.* A derivative 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 derivatives are removed from the store and its bought, but unfinished derivatives are refunded to the buyer.
12. *Completion.* After *maxTurns* rounds, nothing can be bought but raw materials are still delivered and finished until all outstanding obligations are met.

1.3 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 derivative d, we archive $create(p, d)$.
2. When a player p buys a derivative d from *seller*, we archive $buy(seller, p, d)$.
3. When a player p delivers raw material r for derivative 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 derivative 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 derivative that was bought previously.

1.4 Security Concern

It must be difficult for the players to cheat, which is a so called *non-functional* requirement. Currently, we do not implement this requirement, to avoid an overload, but in principle we should. It is not a good idea to add security as an after-thought.

2 Specialization

2.1 For CNF

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

For this formulation, $Typ = \{r1, r2\}$ where $r1$ and $r2$ are *clause types*. A clause type is a pair (l, p), where p is a set of positive literals with l elements. R is the set of weighted CNF-formulas of type Typ, where each clause has a positive integer *weight*. O is the set of all assignments for a CNF-formula. 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:

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

2.2 For CSP

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

For this formulation, *Typ* is the set of boolean relations of at most rank 3, of which there are 256. R is the set of CSP(*Typ*)-formulas. A derivative is a set of relations in *Typ* and a price. O is a Boolean assignment J for a CSP(*Typ*)-formula. 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:

$$(maxVars, maxConstraints, maxWeight)$$

3 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 type of a derivative which you buy only by knowing the type. Now Feng and I are delivering 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 managing 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 (your grade). Your real payout, however, is the set of new skills you learn about managing 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.

4 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.

5 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.)

Acknowledgements: Milena Georgieva Dimitrova has given detailed feedback on the requirements and she has written the previous version of the requirements.