1 Extending CGSynch to your own ruleset

This document will go into depth into numerous implementation details, to explain how CGSynch can be extended for analyzing different rulesets for Alternating and synchronized games. Unlike CGSuite, there is no built-in scripting language that can be used to define these new games, and new rulesets have to be coded directly in `C++`. This is not an easy process, as CGSynch was not made with this requirement in mind.

After adding a new ruleset, CGSynch will have to be recompiled. This should be done with `C++-20` or higher preferably with the flag `-O2` enabled. A `CMakeLists.txt` is included with the code that should do this automatically.

Adding a new ruleset consists of four steps:

- Creating a data structure and database for a position of this ruleset;
- Creating a class containing the logic and rules of this ruleset;
- Adding the created class to the user interface.

We will go over each of these and describe in detail what needs to be implemented. In this section, we will assume the name of your ruleset is `[ACombGame]`. For an example of how to do this, one can look at any of the rulesets already implemented.

1.1 A Data Structure and Database for Positions

For the data structure that represents a position of your game, you choose one that is most suitable for your ruleset. The only requirement is that the data structure is hashable, as it is used internally as a key in the database, which uses a hashmap. It is possible to make your class hashable by extending the `std::hash` function to it. To conform to the type naming in CGSynch, it is advised to `typedef` your data structure to `[ACombGame]Position`.

Next, create the database for storing games of this ruleset. For this, simply put the macro `CreateDatabase([ACombGame]Position, [ACombGame], [databaseName])` at the top of the `.cpp` file that will also contain the logic of your new ruleset. It is advised to use `[ACombGame]database` with the first letter in lowercase as the name of your database. This creates a database for this ruleset and initializes its static member variables so it can be used.

1.2 Logic for the Ruleset

In order to create a new ruleset, you need to create a new `c++`-class `[ACombGame]`, that inherits from the class `AbstractGame<[ACombGame]Position>`. This class requires six functions to be added.

First, it needs a constructor that only takes a `[ACombGame]Position` as an argument. This is relatively straightforward and should just construct a game with the given position.

Secondly, it needs a function `[ACombGame]Position getAnyTransposition() const`. This function should simply return the position of the game. This function will be called by the database when we check whether this position already exists in the database.

Thirdly, it needs a function `std::unordered_set<[ACombGame]Position> getTranspositions() const`. This function should return a set of all possible transpositions of this position that are equal.

All these positions will also be added to the database at the same time, and allow for also finding transpositions of this position. If calculating this is difficult or time-consuming, it could be better to simply return the current position or only this and a few other transpositions that can easily be found.

Fourthly, it needs a function `void exploreAlternating()`. This function should include the logic for playing this position as a combinatorial game. It should set the member variables `leftOptions` and `rightOptions` to contain the `GameId`'s of all left and right positions that can be reached in a single move. It should not recursively determine their left and right positions as well, only its direct left and right options. Adding a game to the database and getting the `GameId` of a game can both be done by calling the member function `getOrInsertGameId` of the database created in the previous step. This will automatically check if a game isomorphic to this game is already in the database, and if it is, return the ID of that game. Otherwise, it will add it to the database and return its new ID. Lastly and importantly, if all direct options of this position could be determined, the member variable `alternatingExplored` should be set to true. This makes sure we do not call this function twice, and instead just use the results that were saved in `leftOptions` and `rightOptions`.
If you are not interested in analyzing alternating games from this ruleset, simply leaving the function empty suffices.

Fifthly, it needs a function `void exploreSynched()`. This function should include the logic for playing this position as a synchronized game. It needs to set the member variable `synchedOptions`. This amounts to setting the number of Left and Right options and setting `synchedOptions.matrix` to contain the `GameID`'s of all positions that can be reached by both Left and Right making a single move. Again, afterwards, the member variable `synchedExplored` should be set to true to indicate that all options have already been explored.
If you are not interested in analyzing synchronized games from this ruleset, simply leaving the function empty suffices.

Lastly, it needs a function `bool determineDecidedSynchedValue()`. This function should determine whether this position is a decided position, and if so, determine its value. If it is not a decided position it should return `false`. Once the value of this position has been calculated in a way that is sensible for this ruleset, the function `getDecidedGameWithValueId` of the `SGDatabase` can be called to get a `SynchedId` with the correct value that should be set to the `synchedId` of your game. Afterwards, `true` should be returned. Again, if you are not interested in analyzing synchronized games from this ruleset, simply leave this function empty.

Additionally, two other functions can be overriden: `bool tryToDetermineAlternatingId()` and `bool tryToDetermineSynchedId()`. Whenever we want to find the Alternating or Synchronized Abstract Game corresponding to this position, these functions are always called first. They are used to determine and execute a shortcut for this position. If they return `false`, it is assumed no optimization was possible, and either `explore` function is called instead. If they return `true`, it is assumed that either the member variable `alternatingId` or `synchedId` is set to a valid Abstract ID. These IDs can be obtained by directly calling member functions on the `CGDatabse` and `SGDatabase` respectively.

Implementing all these functions allows for determining the game tree of a position from this ruleset. The only thing left is to make it possible to create such positions in the User Interface.

1.3 Adding to the UI's

As the interface is based on strings the user enters, we of course need a way to convert a string the user enters into a position for the game that is being added. For this we require a function, usually called `create[ACombGame]position`. By default this function should have the following header: `[ACombGame] create[ACombGame]position(std::string input);`.

Next, the grammar of either the alternating or synchronized user interface needs to be changed to allow for this input to be made. For this, you need to go to the file `SpiritParser.cpp` in the relevant directory, `AlternatingUI` or `SynchedUI`. Here, code needs to be written to do the following five things.

1. Add a lambda that can be called to turn a string into an `[ACombGame]&` object, using the `create[ACombGame]position` function;
2. Add a lambda that calls `getAlternatingId` on an `[ACombGame]&` object;
3. Add a spirit rule to define the boost spirit object that represents an `[ACombGame]`;
4. Add a line to the grammar for reading in a position of `[ACombGame]` using the lambda from step one; and
5. Add a line to the grammar for converting a `[ACombGame]` to an abstract game that calls the lambda from step two.

For more information, we advise taking the current implementation for `Push` and `Hackenbush` as an example and reading the documentation for Boost Spirit (`https://www.boost.org/doc/libs/develop/libs/spirit/doc/x3/html/index.html`).