Documentation of the JSettlers system and the STAC extensions – v. 5

Mihai Dobre
Markus Guhe

26th June 2017

This is a working document describing parts of the JSettlers system, the most important changes we made to it in the STAC project and other (often random) remarks. It is not meant as an exhaustive description or documentation, and the level of description varies between high level and fine technical detail.

Descriptions of algorithms were partly generated by extracting source code documentation. The flow diagrams of the offering methods in §7 were provided by Verena Rieser.

Contents

§ 1 GENERAL NOTES

`JSettlers` is based on the thesis by Thomas (2003), which is still the main reference for the overall system architecture and major design decisions. We are using the enhanced sources from the JSettlers2 project, which are available on sourcefourge (`jsettlers2.sourceforge.net`). We branched out from version 1.1.12 and have not reintegrated any of our modifications into the current project.

On the top level, `JSettlers` consist of a game server (`SOCServer`) to which players connect with their clients, cf. figure 1 in §4. For the server or the other game clients it makes no difference whether a client is a human or an agent. The main client classes are `SOCPlayerClient` for a client providing an interface for a human player and `SOCRobotClient` for agents. An agent in `JSettlers` is called a *robot*.

A robot can have multiple robot 'brains', one for each game the robot is playing. A brain is modelled in the class `SOCRobotBrain`; this is the core of the AI. `SOCRobotBrain` makes use of two other big classes: `SOCRobotDM` (`DM`: Decision Maker) picks what the agent will want to build next and `SOCRobotNegotiator` handles trade interactions.

`SOCRobotClient` handles all messages coming from the server (just like the interface for human players interprets the messages from the server) and sends any messages that require a response or an action to the `SOCRobotBrain` for further handling.

Robots have parameters, specified in `SOCRobotParameters`:

- `maxGameLength`: the maximum game length
- `maxETA`: the max eta
- `etaBonusFactor`: the eta bonus factor
- `adversarialFactor`: the adversarial factor
- `leaderAdversarialFactor`: the leader adversarial factor
- `devCardMultiplier`: the dev card multiplier
- `threatMultiplier`: the threat multiplier
- `strategyType // SOCRobotDM.FAST_STRATEGY or SMART_STRATEGY`
- `tradeFlag`: the trade flag: Does this robot make/accept trades with players? (1 or 0)

We do not make use of these robot parameters in STAC (i.e. they are set to default values), but have our own set of parameters (see below). Of the `strategyType` parameter, we always use the SOCRobotDM.FAST_STRATEGY, because simulations at the start of the project indicated that the minimal improvements of the smart strategy did not justify the increased simulation time for a game by at least one order of magnitude.

§ 2 STAC EXTENSIONS

In addition to this original design, there are a number of major additions in our STAC version, the most notable of which are summarised here (in no particular order).

SIMULATION ENVIRONMENT. The simulation environment provides the means to run multiple games between agents of different types. A sequence of simulations is specified in the `config.txt` file. Logging output is optional (slow but useful for debugging, directories `logs_client` and `logs_server`), and results are recorded automatically (directory `results`), cf. §3 for more details.

The simulation system uses a 'robot factory' (`SOCRobotFactory`) to 'manufacture' the robots required for a simulation.

REPLAY UTILITY. The `SOCReplayClient` makes it possible to replay games from log files in the soclog format.

The package *soc.server.database.stac* contains the required resources to store the game play (excluding chat negotiations) and some `JSettlers` evaluations of the current game state (etw, etb as described later) into a Postgresql database. This can be used to create feature vectors for describing a game state or action (e.g. the *representation* package).

PLAYER INTERFACE. The player interface was modified to meet the requirements for data collection. The standard version we are now using for STAC is to use chat negotiations instead of the original visual trading interface, cf. §9. More precisely, human players cannot use the visual trading interface to make or accept trade offers, but it is still used as a visual aid that is being displayed when robot players make a trade offer.

STAC VERSIONS OF ROBOT CLASSES. To keep the original agent as a baseline for future comparison and to make a distinction between the original and our improved systems, we created STAC-specific versions of the robot classes, which use the prefix `Stac` instead of the usual `SOC`, unless the package name contains the word *stac*. Note that this is/was not always possible (meaning: too cumbersome), so there are some changes to the original `SOC` versions of classes as well. Also, the *smartsettlers* agent (Szita, Chaslot & Spronck, 2010) and the MCTS agent which includes trades have their own brain and factory definitions.

A usual implementation pattern we are using is to insert an additional layer in the form of an abstract Java class which is then extended/overridden by a `SOC` version and a `Stac` version. Classes which do not hold any information and whose instances can be used anywhere in the code (i.e. are completely public) implement an interface (e.g. `SOCRobotFactory`).

PARAMETRISATION. In order to easily test different types of robots, we parameterised the relevant changes/enhancements to the robots. Parameters are defined in the class `StacRobotType` and `MCTSRobotType` for the MCTS agent respectively. Parameters can have integer or double values. For a simulation, they are specified in the file `config.txt`.

IMPROVED INITIAL PLACEMENT OF PIECES. The `StacRobotDM` uses an improved placement of pieces during the setup phase. This is a marked improvement over the original `JSettlers` strategy. The robot we now call *original* already uses this improvement.

DECLARATIVE MEMORY. `JSettlers` does not use a principled approach to modelling knowledge, which is particularly problematic for the implementation of the agent/robot. Throughout the system, knowledge representation and decision making is tightly interwoven in long sequences of heuristics. This makes it very hard to understand how `JSettlers` represents knowledge, how decisions are made and how different knowledge results in different decisions.

Additionally, STAC is concerned very much with preferences and beliefs. For this reason, we moved the representation of 'declarative' knowledge to its own class. This distinction is based on the one traditionally made between declarative and procedural memory/knowledge usually found in AI and cognitive systems. It is clear that there is no sharp distinction between these knowledge types. For our purposes here and at the current state of development, we understand declarative knowledge to encompass:

- beliefs about opponent resources
- beliefs about what opponents are selling (`JSettlers`: `is-selling`)
- beliefs about the opponents next moves/build plans
- information about opponents behaviour
- information about our behaviour towards other players
- memory of past trade offers (trade offers made by the agent in the current turn)
- memory of the agent's potential build plans
- dialogue history (about to be added)

This knowledge is now represented in `StacRobotDeclarativeMemory`, and all uses (store and retrieve operations) of the types of knowledge listed above are now calling this class. The observable information as well as information on the players' data (`SOCPlayer`) and trackers (`SOCPlayerTracker`) can now also be accessed via this class, however the actual data objects are still maintained in the `StacRobotBrain`. Note that the `SOCGame` object stored in the brain also gives access to information which is modelled in the declarative memory (e.g. the resources in the players hands), so when working on the source code, please avoid accessing this information directly.

In an extension to the JSettlers system, we also added an interface to the ACT-R cognitive architecture (in a Java reimplementation: jACT-R) to store these beliefs (build plans not included). StacRobotDeclarativeMemoryACTR subclasses StacRobotDeclarativeMemory for this purpose (the interface to the rest of the system remains unchanged).

PARTIAL AND DISJUNCTIVE OFFERS. A major difference between JSettlers and the way people negotiate is that JSettlers can only make complete trade offers (*I want to get 1 ore for 2 wheat from player-A or player-B.*), whereas people usually make partial trade offers (*I need ore*) and sometimes disjunctive offers (*I need ore or wheat for sheep*). The systems now allows partial and/or disjunctive offers, and the agents can handle them. There are also agents that make such offers, even though their performance suggests that they must be improved.

NEGOTIATION. Because the original negotiation methods are difficult to understand and modify, we created a new negotiator, which is largely a reimplementation of the original methods, cf. §8. This new negotiator is now the standard negotiator being used by the robot. It mainly defines a new offering strategy (method makeOffer), which does not quite reach the same level of performance as the original method. Nevertheless, performance only suffers slightly, and this is outweighed by the much clearer implementation.

DIALOGUE MANAGER. We implemented a first version of a symbolic dialogue manager (StacDialogueManager), which provides a framework for dialogue managers to be developed in STAC. The manager provides two functions: trading and announcements of various information, both done via the game's chat interface.

Robots use the dialogue manager if the static flag chatNegotiations in the StacRobotBrain is set. This can be done using the ChatNegotiations field in the config.txt file (§3) or by checking the corresponding box when starting a game via the SOCPlayerClient. Sharing/announcing resources/plans or listening to such announcements can be enabled by defining the corresponding robot types. See §9 for a detailed description of the dialogue manager's current capabilities.

MONTE CARLO TREE SEARCH. The full MCTS algorithm extended to the complete action space is implemented in a separate project and imported via an external library (MCTS-1.0.jar in lib folder). A Flat-UCT implementation is included in the package *soc.robot.stac.flatmcts*. It can integrate domain knowledge by seeding into the tree nodes. It can only be applied to the initial placement and robber action. The SmartSettlers implementation is present in the *originalsmartsettlers* package (Szita et al., 2010).

SAVING AND LOADING GAMES. Games can be saved and loaded in specific states. This is useful for running simulations from a specific game state or to test different robot types in a controlled environment. Games can be loaded at any time during or before starting a game, but only the current player can save the game. There are some additional limitations:

- A human player cannot save the game when a second window pops up during the game (i.e. after playing a Monopoly card and before choosing the resource to monopolise; after playing a Year of Plenty card and before choosing the two resources to 'discover'; after placing the robber and before choosing the player to rob and before choosing resources to discard).
- Saving/loading during trading is not supported due of the complex nature of the interaction.
- For a human player, if a trade was initiated during the turn, either cancel or clear buttons need to be pressed before being able to save again, even if the trade was completed.
- A robot cannot always replace a human player, e.g. after a piece was bought or the Road Building development card was played the robot won't have a build plan, let alone a build plan for the specific piece.

SPECIFIC BOARD CONFIGURATIONS. Part of saving a game state is that the configuration of the board is saved as well. Such a saved board configuration can be used when a new game is being started by specifying the `BoardLoad` parameter in the `config.txt` file, cf. §3.

DEBUG UTILITY. `JSettlers` has its own debug implementation in class `D`, which prints out any information to the standard output stream. Depending on the package used (*soc.debug* or *soc.disableDebug*), one can deactivate/activate debugging for each class individually. This has now been extended to print out the debug messages based on the level of importance of the message. There are four debug levels: `INFO`, `WARNING`, `ERROR` and `FATAL` which are used with the corresponding methods (e.g. `debugWarning`). If debug is enabled, setting the level to any will only print the messages of the same importance or higher. Note that `FATAL` represents exceptions, which the original system was already handling. These do not always kill/stop the server despite the name, but are always very harmful to the game that caused them. `ERROR` level messages may later result in a `FATAL` message so these shouldn't be ignored. `WARNING` messages represent issues which are accounted for and handled, but for which we may want to track the number of occurrences.

REINFORCEMENT LEARNING. The *soc.robot.stac.learning* package contains the Jedi Mind Trick implementation as well as a Q-learning agent which can be trained via the JSettlers framework.

§ 3 HOW TO RUN A SIMULATION IN JSETTLERS

The main class for running agent-only simulations is `Simulation`, which is in package *soc.robot.stac.simulation*. By default, it looks for the file `config.txt`, which specifies the specifics of the simulation, in the *resources* package, but a different file can also be specified as parameter.

Please note that, where applicable, the parameters in `config.txt` also have an effect when `JSettlers` is not run for a simulation but with another main class. For example `SOCPlayerClient` can be run to play local games where the parameters have an effect.

From the command line, a simulation is usually started with:

```
java -cp STACSettlers.jar soc.robot.stac.simulation.Simulation [config-name]
```

The structure of a config file is as follows:

```
1  Games={number of games}
2  Log=[true|false]
3  NoShuffle=[true|false]
4  PlayerToStart=[-1|0|1|2|3]
5  Load=[true|false]
6  FolderName={String}
7  NoTurns={maximum number of turns}
8  ChatNegotiations=[true|false]
9  BoardLoad=[true|false]
10 UseParser=[true|false]
11 Debug=[true|false]
12 ForceEndTurns=[true|false]
13 FullyObservable=[true|false]
14 CollectValueFunctionApproxWithID={specify a new table id}
15 CollectFullGameplay=[true|false]
16 ~
17 {SimulationName}
18 Trades=[true|false]
19 Control={control parameters, applied to all agents, separated by |}
20 VarParam={parameter for all agents}|{start value}|{step}|{number of steps}
21 RoundRobin
22 Agent={N},{name},[stac|jsettlers|random|mcts|ss],{parameters, separated by |}
```

The lines have the following function

1. Specify the number of games per simulation.
2. Turn logging on or off. (Output is written to directory `logs`.)
3. Boolean value; `false` if players' positions can be randomised, `true` if the players initial position is kept for all games in a simulation.
4. Player starting the game, specified by an integer corresponding to the seat position (`0`: blue, `1`: red, `2`: white, `3`: orange, `-1`: random).
5. Boolean value describing if each of the simulations should start from a previously saved state; `true` also requires `FolderName` and that the saved

files be placed at the defined path.

6. String value containing the name of the folder under the `saves` folder containing the saved game. If no game is loaded, this parameter should be omitted.
7. Integer value defining the maximum number of turns for each game in the simulation or 0 if the games should resume until the victory conditions are met.
8. Specify whether robots and players use chat negotiations.
9. Decide whether to load a saved board configuration. Use this parameter only if it is a normal start of game. It can be set to `true`, which means the file containing the bytes of the `SOCBoard` object is read from `saves/board/soc.game.SOCBoard.dat`.
10. Call the Toulouse parser each time a non-robot player (i.e. a human player) sends a line of chat to the server. (This means, this option should have no effect in simulations but only when using the `SOCPlayerClient` class.) `JSettlers` does not contain or start the parser and simply assumes it exists and can be accessed via the usual TCP port.
11. Turn debug on or off.
12. Specify if the robot's turn should be ended forcibly if they take too long to decide on an action.
13. Fully observable game or the player hands and development cards are hidden.
14. Specify an id for a table to collect the state values of the heuristic players.
15. Specify if the simulation games should be stored in the database.
16. Separator starting a new simulation block. A config file can specify multiple simulation blocks, each of which specifies one simulation. The simulation blocks are executed in the order in which they are specified.
17. The name of the simulation. It is also used to name the log files and the directory in which the results are stored (these are subdirectories of `results`).
18. Turn trading on or off for all agents. (Note that there is a `NO_TRADES` robot parameter to switch off trading for individual agents.)
19. List of robot parameters (as specified in `StacRobotType`) that are applied to all agents. Multiple entries are separated by `|`.
20. A variable robot parameter that is applied to all agents. Three values specify the start value for the parameter, the magnitude of each step and the number of steps.
21. A 'round robin' run of simulations. This assumes that the first agent listed in the block is the control/baseline agent and there are four experimental agents. The round robin consists of these simulations:
    - All experimental agents against each other.
    - All combinations of 3 control agents vs 1 experimental agent.
    - Each combination of 3 experimental agents vs the control agent

    Note: Round robin and variable parameters are mutually exclusive.
22. Specification of one type of agent participating in the simulation. There can be multiple lines of this type. The four, comma-separated values are:
    i. The number of agents of this type.

ii. The name of the agents of this type.
iii. The Java class implementing the robot factory that is being used to create the robots. Possible values are:
   - `stac` for the StacRobotFactory
   - `jsettlers` for the SOCDefaultRobotFactory
   - `random` for the SOCRobotRandomFactory[1]
iv. List of robot parameters (specified in StacRobotType). Multiple entries are separated by `|`.

If any of the parameters in the first block are not specified, than these are set to the default values as specified in Simulation class. Usually, robot parameters (item 19) are used as boolean values, i.e. at the relevant point(s) in the code it is checked whether the parameter is set. Depending on the parameter definition, it can also specify an integer or double value, which are separated by a colon. For example, the `MIN_VP_TO_TRY_LR` parameter modifies the standard building strategy so that it only starts considering trying to get the Longest Road at a different point in the game: usually the robot considers Longest Road only if it has at least 5 Victory Points. Specifying `MIN_VP_TO_TRY_LR:3` has the effect that the robot makes this consideration when it has 3 Victory Points or more.

`NoShuffle` and `PlayerToStart` can be used together to control the position and order of play, allowing to test how these affect the performance of robots (e.g. if the agent being tested is defined first after a `~` separator, then `NoShuffle` can be set to `true` and `PlayerToStart` to `0` in order to have the agent start every game in the simulation batch).

The fields defined in the preamble (i.e. the part before the first `~`) have default values set inside the Simulation class, so these can be omitted from the config file (however the separator needs to be on the first line of the file).

Here is an example for a config file that can be used to test the new negotiator against the 'original' agent. It runs a standard simulation of one modified agent against three original (STAC) agent. Note that the default is to use the new negotiator (cf. §8).

```
Games=10000
Log=false
NoShuffle=false
PlayerToStart=-1
Load=false
BoardLoad=false
NoTurns=0
Debug=false
~
NewNegotiator
Trades=true
```

[1] This factory creates robots using StacRobotBrainRandom, which are robots that randomly choose a ('best') build plan but use the standard trading strategy.

```
Control=ASSUME_OTHER_AGENTS_USE_OUR_STRATEGY|ALWAYS_ASSUMING_IS_SELLING
Agent=1,newNegotiator,Stac,ORIGINAL_ROBOT
Agent=3,original,Stac,ORIGINAL_ROBOT|USE_OLD_NEGOTIATOR
```

If a practice game is launched by running `SOCPlayerClient` (1 human player vs 3 robots), the type of robots filling the spaces can also be defined in the config file. In this case, only the `Agent` fields are read and the other parameters in the file are ignored, except the `UseParser` parameter.

§ 4 THE ROBOT CLIENT

The `SOCRobotClient` manages the connection to the server. It creates the robot brain, and in particular, it creates one brain for each game.

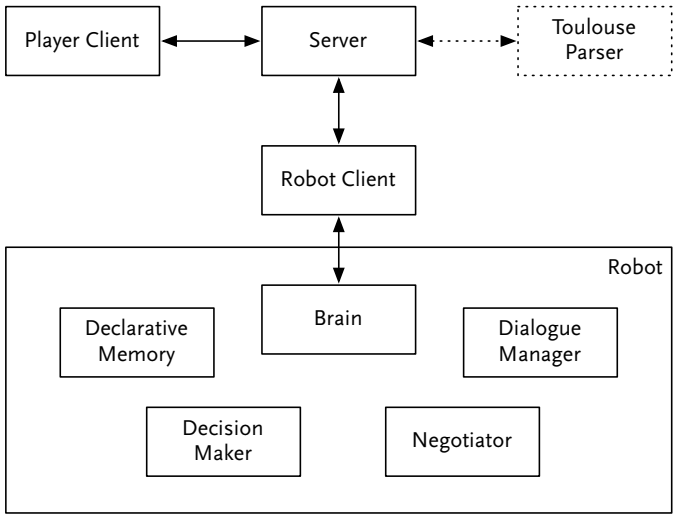

Figure 1: Overall architecture of the `JSettlers` system.

For a simulation, i.e. if the same robot plays multiple games in a row, the robot client is persistent, i.e. it is created at the start of the simulation and then stays alive till the end of the simulation, and for each game the robot plays, it creates a new robot brain.

In a simulation, each robot client only plays one game at a time, i.e. the next game only starts after the previous one has ended.

§ 5 THE ROBOT BRAIN

The main `run` loop in `SOCRobotBrain` handles sending and receiving messages to/from the server, planning what to build next (`SOCRobotDM`) and whether to

trade (given the next piece to build, SOCRobotNegotiator considers possible trades and decides whether to make a trade offer, accept a trade offer or make a counteroffer).

The Stac robot has two additional classes: StacRobotDeclarativeMemory and StacRobotDialogueManager. The declarative memory contains the robot's (observable and unobservable) declarative knowledge. We introduced it for two main reasons. Firstly, the JSettlers robot stores knowledge in different classes (not just brain, negotiator, decision maker but also its own SOCGame object among others) and does not provide a unified access, which means it is all but impossible to retrieve all of the robot's knowledge, for example, to make an inference based on the robot's knowledge state. Secondly, the declarative memory facilitates modelling memory processes (e.g. forgetting or associative strength), which has already been used for creating a version of the robot where parts of the declarative memory have been substituted with an ACT-R model.

The dialogue manager handles messages that the clients exchange via the chat interface, see §9 for details. These messages fall into three main categories: announcements, e.g. sharing of the current build plan, embargo persuasion attempts and strings representing TradeMessage objects.

SOCRobotBrain uses SOCPlayerTracker to track possible building spots for itself and other players. This means that the player trackers only track observable information. Dissertation excerpt:

> When a player places a road, that player's PlayerTracker will look ahead by pretending to place new roads attached to that road and then recording new potential settlements [and their roads] ... The PlayerTracker only needs to be updated when players put pieces on the board ... not only when that player builds a road but when any player builds a road or settlement. This is because another player's road or settlement may cut off a path to a future settlement. This update can be done by keeping track of which pieces support the building of others. (Thomas, 2003, p 97–98)

For a legible overview of the data stored in a SOCPlayerTracker, use method `playerTrackersDebug(HashMap)`.

Method `buildOrGetResourceByTradeOrCard` either asks to build a piece or uses trading or development cards to get resources to build it. It examines the `buildingPlan` (of type SOCBuildPlan), which is a stack of SOCPossiblePiece objects, for the next piece wanted. `buildOrGetResourceByTradeOrCard` is called when these conditions are all true:

- `gameState` is `SOCGame#PLAY1` or `SOCGame#SPECIAL_BUILDING`
- `waitingFor...` flags all false (`waitingForGameState`, etc) with the possible exception of `waitingForSpecialBuild`

- `expect...` flags all false (`expectPLACING_ROAD`, etc)
- `waitingForOurTurn` is false
- `ourTurn` is true
- `expectPLAY` is true and `counter < 4000`
- `buildingPlan` is not empty (`!buildingPlan.empty()`)

It may set any of these flags:

- `waitingForGameState`, and `expectWAITING_FOR_DISCOVERY` or `expectWAITING_FOR_MONOPOLY`
- `waitingForTradeMsg` or `waitingForTradeResponse`, `doneTrading`
- `waitingForDevCard`, `waitingForGameState`, `expectPLACING_SETTLEMENT` (etc).

Algorithm:

- If we have a Road Building card and the top item of the `buildingPlan` is a road building plan, i.e. the next two pieces to be built are roads, try to build them using the card. (Check if the agent already unsuccessfully tried to build in that location; if not, try to build there.)
- If the top item is a plan to build another piece, try to get the required resources.
  - If the agent has a Year of Plenty (Discovery) card and needs at least 2 resources, consider playing it.
  - If the agent has a monopoly card, consider playing it.
  - If we're not done with trading, try to make a trade offer to get the resources needed for the build plan.
  - If we cannot find a suitable trade offer, trade with the bank or a port.
  - If we have the necessary resources, build the target piece.

Method `tradeStopWaitingClearOffer` ends the wait for responses to a trade offer and remembers other players' responses. This is called, i.e. the agent stops waiting for responses to a trade offer, when the internal counter has exceeded 10,000.

THE RUN LOOP. While the brain is 'alive' (i.e. the `alive` flag is true), the `run` method executes an infinite loop that reacts to messages sent from the server and initiates planning and game actions.

Note that the description below does not consider the handling of 6-player games.

- Read the next message from the event queue and count it (our crude internal clock).

- Check if it's a `GAMECOPY` or `LOADGAME` message and suspend the brain until the process is finished.
- Check if we're waiting for a response to our trade offer and have reached the timeout.
- Check whether we're waiting for a `GAMESTATE` message; resend the request if the timeout is reached.
- If we received a `GAMESTATE` message, update the game state to the new value; if the message starts a `PLAY1` phase of turn (i.e. after rolling the dice), execute the "start-turn" actions (e.g. start of chat).
- `FIRSTPLAYER`: Set the player starting the game.
- `SETTURN`: Set the player whose turn it is.
- `TURN`: Start a turn.
  - Set `expect...` flags to `false`.
  - Determine whether this robot is trading and reset trading flags (we're not in the middle of a negotiation).
  - Reset the is-selling information in the negotiator.
  - Reset the past-trade-offer information in the negotiator.
  - Reset our build plan.
  - Reset the target pieces in the negotiator.
- `GAMESTATS`: Update the learner with the results of a game.
- If we are the current player, note that it is our turn.
- `TURN` and it is our turn: Note the we don't wait for our turn any more and reset the records of failed build attempts.
- `PLAYERELEMENT`: Update a player's amount of a resource or a building type. If this during the `SOCGame#PLAY` state, then update the `SOCRobotNegotiator`'s is-selling flags. Only the game data is updated, nothing brain-specific.
- `RESOURCECOUNT`: The server informs us about how many resources a player has in total. Check whether this matches our model (beliefs) and correct if necessary.
- `DICERESULT`: The result of a dice throw. Update the current dice result information in our game model.
- `PUTPIECE`: A piece has been put on the board; update our internal information.
- `CANCELBUILDREQUEST`: The server informs us that we cannot build a piece in the location we requested. So we have to revert and come up with a new plan.
- `MOVEROBBER`: The robber has moved, so update our internal state. This does not handle the stealing.
- `MAKEOFFER`: If another player makes an offer, that's the same as a rejection, but still wants a deal. Call `considerOffer(SOCTradeOffer)`, and if we accept, clear our build plan so we'll replan it. Ignore our own `MAKEOFFER`s echoed from server.
- `CLEAROFFER`: Clear the offer for the specified player or all players.
- `ACCEPTOFFER`: If we're waiting for a trade response and it is our offer that

is being accepted, note this and determine whether to keep waiting for responses. Update negotiator accordingly.

- `REJECTOFFER`: Handle a rejection to an offer. Assume that the rejecting player is not selling the resources asked for and update is-selling accordingly. If it is our own offer that is rejected, also check whether we have a response from all players we made the offer to and whether we believe the rejecting player may want another offer.
- `DEVCARDCOUNT`: Update the number of development cards remaining in the deck.
- `DEVCARD`: A player is drawing or playing a development card. Update our information.
- `SETPLAYEDDEVCARD`: Set the flag which says if a player has played a development card this turn.
- `POTENTIALSETTLEMENTS`: Receiving a list of potential settlements. Update out information.
- `GAMETEXTMSG`: Treat chat messages from opponents in the `StacDialogueManager` and send a response if appropriate.
- On our turn, ask the client to roll dice or play a knight card (if we have one and the robber is on one of our hexes); on other turns, update flags to expect dice result. Clear `expectPLAY` to false. Set either `expectDICERESULT`, or `expectPLACING_ROBBER` and `waitingForGameState`.
- If we have to place the robber, place it now.
- If it's our turn and we're waiting to pick our cards for having played a Discovery (Year of Plenty) card, tell the server which cards we want.
- If it's our turn and we're waiting to pick resources after having played a Monopoly card, tell the server which resource to monopolise.
- If we're waiting for a trade message and received a `GAMETEXTMSG` from the server, check whether this informs us of the trade or the fact that we wanted to make an illegal trade. Stop waiting for a trade message. The trade could be a trade with the bank/a port or us having accepted a counteroffer.
- If we're waiting for a development card and got the corresponding `GAMETEXTMSG` from the server stop waiting for the development card.
- If there is nothing else to do or wait for, it's time to decide to build or take other normal actions.
  - If it is our turn, and we haven't played a development card yet, and we have a knight, call the decision maker to decide if we should play it for the purpose of acquiring the Largest Army.
  - If it is our turn, and we don't have a build plan, and we haven't given up building attempts in this turn (because we made too many illegal building requests that the server rejected), make a build plan.
  - If it is our turn, and we are not waiting for the robber to be placed, and we have a build plan, execute the plan or try to make a trade for the required resources.
  - If it is our turn, and we're not expecting the server to inform us that

one of our actions is completed and we're not waiting for a trade or development card action, end the turn.
  - If we decided to end our turn, check if `TRY_N_BEST_BUILD_PLANS` is set and try building the next plan in our possible plans list, until we have tried N plans and then end our turn.
- If the server has told us that it's OK to build in the location we want, build there.
- `SETTURN`: Set the current player.
- `PUTPIECE`: Handle putting down of a piece by updating `SOCPlayerTrackers`. Also handles placement of pieces during the setup phase.
- `DICERESULT`: If we're expecting a dice throw, check whether it's a 7 and initiate discarding of resources and (if it's our turn) moving of the robber. Otherwise change to the main phase of the turn.
- `DISCARDREQUEST`: If the current dice result is a 7 and it is our turn, set the flag for expecting to place the robber, or move to the main phase of the turn otherwise. Discard the required number of resources.
- `CHOOSEPLAYERREQUEST`: Choose a player to be robbed. (I'm assuming this happens if the robber is being placed on a hex that touches multiple players' settlements/cities.)
- `ROBOTDISMISS`: Make this robot leave the game.
- `GAMETEXTMSG`: The server sends a `*PING*` message once a second. Treated and counted here.
- If we're waiting for more than 15,000 pings, leave the game and terminate this robot.
- If we cannot decide where to put our initial settlements too often, leave the game and terminate this robot.
- After the 'infinite' loop finishes, clean up by resetting our internal state.

§ 6 GAME STRATEGY (THE ROBOT'S DECISION MAKER)

There are four main game strategies implemented in `StacRobotDM`:

1. `dumbFastGameStrategyOld`. The old implementation of the original strategy.
2. `dumbFastGameStrategy`. Reimplementation of the original strategy that first generates all potential build plans and stores them in the `StacDeclarativeMemory`. This is now the default method.
3. `dumbFastGameStrategyNBest`. A simple n-best strategy: Instead of the best build plans, choose the $b - n$ strategy from the potential build plans. Note that it was intended that $b - 0$ is the same as the original strategy, but playing against 3 originals, we actually get an improvement of about 4.5 percentage points over the original strategy (in both implementations).
4. `smartGameStrategy`. The smart strategy that comes with the original `JSettlers` system. This strategy does not only consider the next build action but tries

to plan further ahead in a smart way.

These 'main' strategies can be modified with many other robot parameters (as defined in `StacRobotType`, e.g. by favouring particular types of build plans.

Per default, all methods use the same scoring functions for the Estimated Time to Build (ETB) and the Estimated Speedup (ES) that are the basis for selecting the best build plan. Guhe (2014) describes a new scoring function where ETB, ES and Estimated Progress (how much closer to winning the game does the player get by executing the build action) can be traded off against each other, effectively enabling the agent to forego a short-term benefit for a long-term gain.

Note that the original `JSettlers` agent did not implement the method for computing the ES for settlements. This is now enabled throughout.

OLD IMPLEMENTATION OF THE ORIGINAL STRATEGY. There are robot parameters that can favour building certain types of pieces with a specified factor, e.g. settlements or cities. This is realised by adjusting the ETB for a build plan of the respective type with the formula $etb - (factor * etb)$. When such parameters are set, choosing a build plan is always performed by using the adjusted ETBs.

The original game strategy is as follows:

- Get the minimal number of VP needed to consider Largest Army or Longest Road (4 by default).
- If we have fewer than or equal this number of VPs on both of these criteria, do this:
  - Score the possible cities and get the one with the highest speedup (ES).
  - Update the scoring of possible settlements.
  - Get the best settlement (including roads required to build), i.e. one with an ETB lower than that of the best city. If best city and best settlement have the same ETB use the ES as a tiebreaker. (If there are enough settlement and road pieces left.)
  - If there is a settlement build plan (including roads), choose it.
  - If there is no settlement build plan, choose the best city build plan if there is one.
  - If there is neither a settlement nor a city to build, buy a development card if there are any left in the deck.
- If we have more than the minimal number of VP, do this:
  - If we're above the threshold, calculate the ETB for Largest Army. Don't go for Largest Army if we already have it or there are fewer cards in the deck than we need knights to get Largest Army. Choose Largest Army for now.
  - If we're above the threshold, calculate the ETB for Longest Road. Don't

go for Longest Road if we already have it. Don't go for Longest Road if we can't establish a path for it or don't have enough Road pieces left. Choose Longest Road if its ETB is less than the best ETB so far (i.e. the one for Largest Army).
- If the ETB for building a city is lower than the best ETB so far: Score the possible cities and get the one with the highest speedup. Choose this (for now).
- Update the scoring of possible settlements.
- Get the best settlement (including roads required to build first) that has a lower ETB than the best ETB so far. If both ETBs are equal, use the ES as a tiebreaker between best city and best settlement. (If there are enough settlement and road pieces left.) If there is one, choose it.
- Depending on the chosen type of piece to build, generate the build plan. Note that in this branch development cards are not bought as a last resort but only as part of the Largest Army build plan.

REIMPLEMENTATION OF THE ORIGINAL STRATEGY. This reimplementation recreates the strategy and the decisions made from the original strategy but uses the potential build plans in `StacRobotDeclarativeMemory`. It is the same in terms of the usual measures of win rate, number of offers, number of successful offers and number of trades.

- Generate all possible build plans with method `generatePossibleBuildPlans`, see below.
- Find the best city build plan.
- Find the best settlement build plan (including required roads).
- If there is a best city build plan, choose it for now.
- If there is a best settlement build plan with a lower ETB, choose it for now. If both ETBs are equal, take the build plan with the higher ES.
- If we're above the threshold for considering Largest Army and there is a plan for getting it, choose it if the ETB is lower than the best ETB so far.
- If we're above the threshold for considering Longest Road and there is a plan for getting it, choose it if the ETB is lower than the best ETB so far.
- If we still have no build plan and there is a plan for getting a development card and the number of our VPs is below both thresholds for getting Largest Army and Longest Road, choose it.

GENERATING POSSIBLE BUILD PLANS. Method `generatePossibleBuildPlans` is the core of the reimplementation of the original strategy and the new n-best strategy. It generates all possible build plans and for each plan computes ETB and ES. ES is really only available for City and Settlement build plans and not for Largest Army, Longest Road or Development Card, but as part of the 'trading off' extension (Guhe, 2014) there are estimates for ES as well as for EP stored with the build plans (where appropriate). All build plans

are stored in StacRobotDeclarativeMemory.

This method does not check for a minimum of Victory Points to build particular pieces (this is done by the decision methods), but takes into account:

- that enough pieces of the required types are still available,
- favouring building particular types of pieces,
- whether we already have Largest Army or Longest Road, in which case no build plan of the corresponding type is generated.

N-BEST STRATEGY. The n-best strategy (`fastDumbStrategyNBest`) also generates all possible build plans (`generatePossibleBuildPlans`). It simply chooses the best-minus-n build plan. Thus, while we would assume that the above two methods also pick the build plan with the lowest ETB, the main difference is that development cards are not treated as a special case but are just treated in the same way as all other types of build plans. Consequently, they are picked much more often (about half of the time, in fact).

- Generate all possible build plans.
- Get all possible build plans from StacRobotDeclarativeMemory.
- Remove the build plans for Largest Army, Longest Road and Development Card if the current VPs are below the respective threshold (below both thresholds to remove the Development Card plan).
- Sort the possible build plans by increasing ETB. For equal ETBs sort by decreasing ES. For multiple build plans with equal ETB and ES the ordering is random.[2] (So, the best build plan is one with the lowest ETB and highest ES, but note the potentially different behaviour resulting from the extensions described in Guhe, 2014).
- Choose the top n-th build plan from the list.

§ 7 NEGOTIATING TRADES (THE ROBOT'S NEGOTIATOR)

Once the robot brain has chosen a build plan (with the help of the decision maker), it checks whether it has the resources to execute the plan immediately. If not, it may decide to try to trade for the resource, which is when the brain calls on the negotiator.

The class SOCRobotNegotiator contains the code to make and process offers. There are four main methods: `makeOffer`, `makeOfferAux`, `considerOffer` and `makeCounterOffer`. SOCRobotBrain.`makeOffer`, which is called as part

[2] It is random in principle, but as the original order of possible build plans in the list is: Settlement, City, Development Card, Largest Army, Longest Road, it is to be expected that this order is retained for these cases.

of the agent's main run loop, makes the call to the negotiator's `makeOffer`, cf. figure 2.

The decision process starts with determining what to build next (the target piece; in STAC, we usually call this the *Best Build Plan*, or BBP; this is done by SOCRobotDM). SOCRobotNegotiator then computes trade offers with this main method: `SOCTradeOffer makeOffer(SOCPossiblePiece targetPiece)`;

- `targetPiece` – the piece that we want to build;
- return value: the offer the robot wants to make, or null for no offer

`makeOffer`'s algorithm is roughly this:

- If we already have the resources needed for building the `targetPiece`, do not make an offer.
- Determine the resources required for the target piece (target resources).
- Determine Best Alternative To a Negotiated Agreement (BATNA); meaning: what trade would we have to make with the bank of a port to get the resources.
- Compute the building speed estimate (uses class SOCBuildingSpeedEstimate) to determine the time it would take us to build the piece (1) with our current resources, i.e. without trading, and (2) with the BATNA.
- Separate resource types into needed and not-needed. Sort groups by frequency, most to least. Start with most frequent not-needed resources. Trade for least frequent needed resources.
- Make a list of what other players are selling (this information is gathered from previous trade offers and is stored in the array `isSellingResource`, see below)
- See if other players are selling resources we need.
- Determine the resources that (1) are required and (2) someone is selling
  - consider offers where we give one unneeded for one needed
  - consider offers where we give one needed for one needed (make sure the offer is better than our BATNA)
  - consider offers where we give two for one needed (make sure the offer is better than our BATNA)
  - consider offers where we give one for one unneeded that we can use at a bank or port (consider giving one unneeded (make sure the offer is better than our BATNA); then consider giving one needed (make sure the offer is better than our BATNA))

The trade offers the agent and the other players make in the current turn are being cached to make the whole process less repetitive. (Thus, although this is no proper cognitive memory model, the agent does have a certain memory function. This information is now stored in StacDeclarativeMemory.)

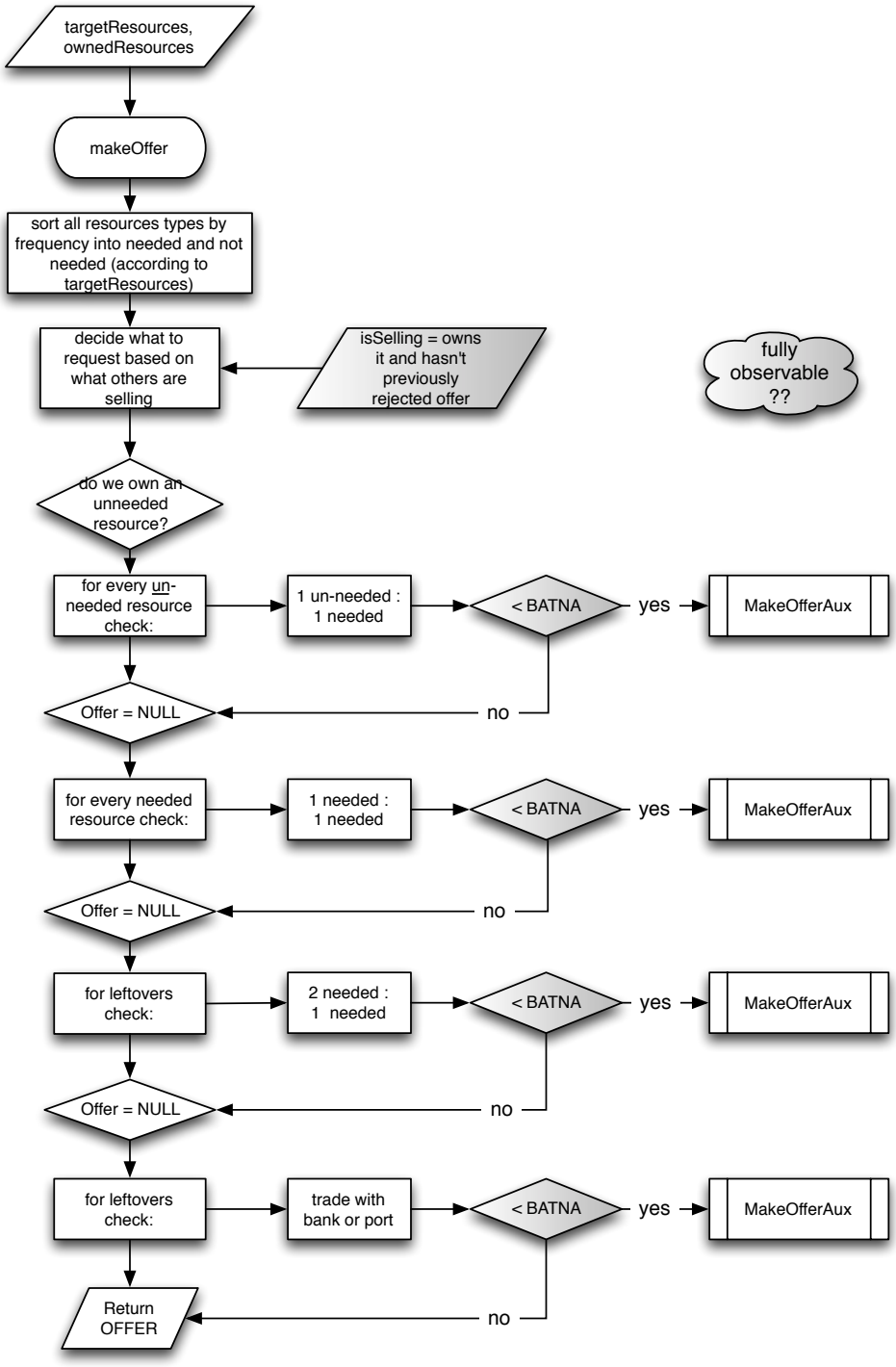

Figure 2: The `makeOffer` method from `SOCRobotNegotiator`.

Offers are only made to players that are selling what the agent is asking for and aren't too close to winning (method `makeOfferAux`, cf. figure 3). Note the for a while now we are disabling the checks with `isSelling` in simulations by setting parameter `ALWAYS_ASSUMING_IS_SELLING`.

Offers are only made if the robot thinks that someone will accept it.

Method `getETAToTargetResources` returns the estimated number of rolls until the robot reaches the specified target given a potential offer.

Method `considerOffer2` decides whether to accept, reject or make a counteroffer, cf. figure 4. It basically accepts an offer if (1) it has the required resources and (2) accepting the offer reduces the building time over the BATNA building time. (This seems to be a more efficient reimplementation of the original method by Robert Thomas.)

Method `makeCounterOffer` decides whether to make a counteroffer, cf. figure 5. This works along the same lines as `makeOffer`.

Method `getETAToTargetResources` computes the estimated number of rolls until the agent reaches the target given a possible offer; parameters:

- `player`: our player data
- `targetResources`: the resources we want
- `giveSet`: the set of resources we're giving
- `getSet`: the set of resources we're receiving
- `estimate`: a `SOCBuildingSpeedEstimate` for our player
- return value: the number of rolls to get the target resources

The agent observes other trade interactions and keeps track of what other players are not willing to trade in the array `isSellingResource`. Values are set by the methods

- `markAsNotSelling`: Called (1) when the agent observes a trade offer by another player: the resources requested by that player he is not selling; (2) when another player rejects a trade offer; (3) when the agent retracts an offer because no player took up the offer in the predefined time.
- `markAsSelling`: Not called at any point. (Resources are marked as 'is-selling' by `resetIsSelling`.)
- `resetIsSelling`: Resets the `isSellingResource` array so that if the player has the resource, then he is selling it. Called (1) as part of handling a `PLAYERELEMENT` message; (2) at the start of each turn; (3) when a `SOCRobotNegotiator` is initialised.

§ 8 THE NEW NEGOTIATOR

Class `StacRobotNegotiator` contains the reimplementation of the negotiator. The overall approach is similar to the changes in the decision maker: generate

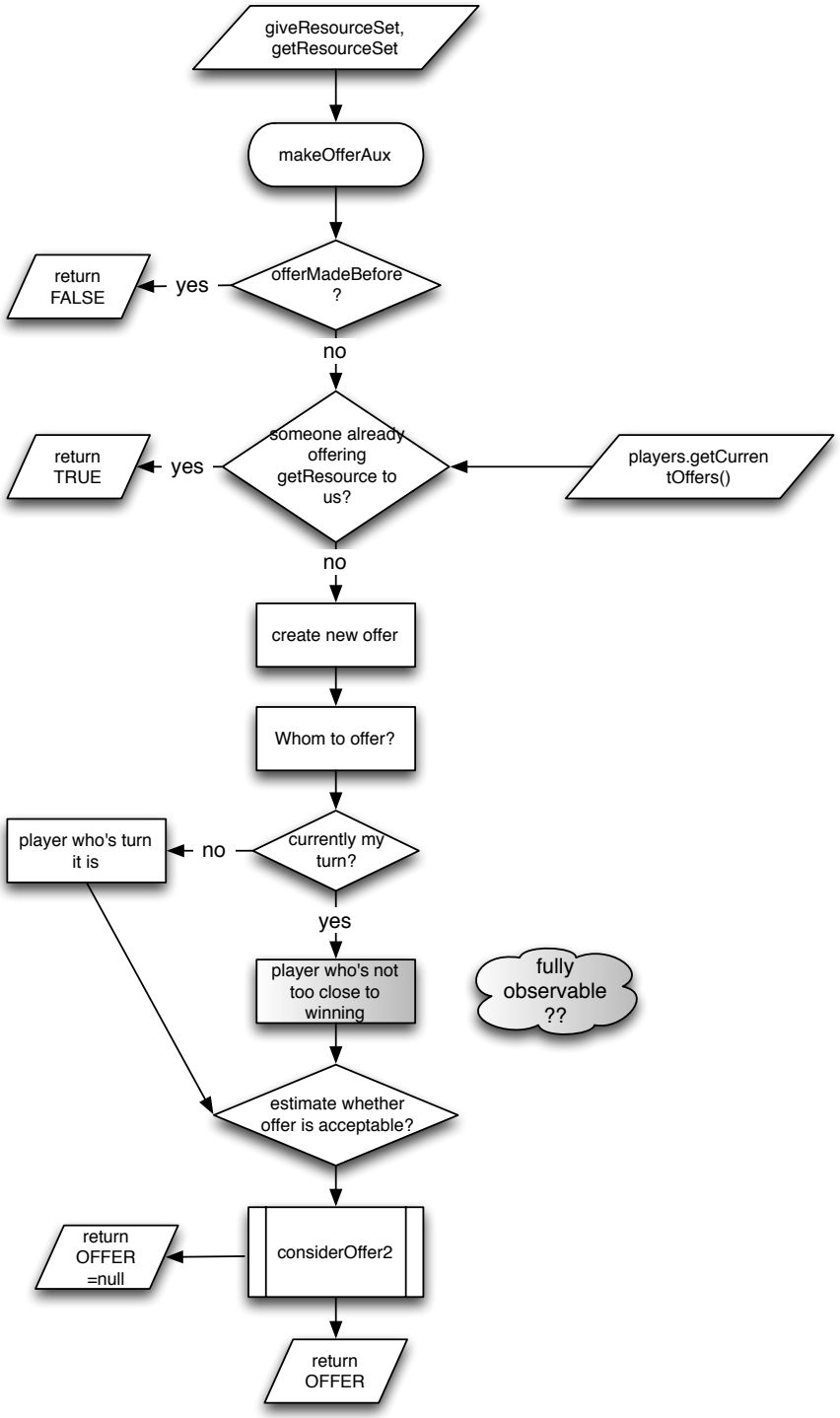

Figure 3: The `makeOfferAux` method from `SOCRobotNegotiator`.

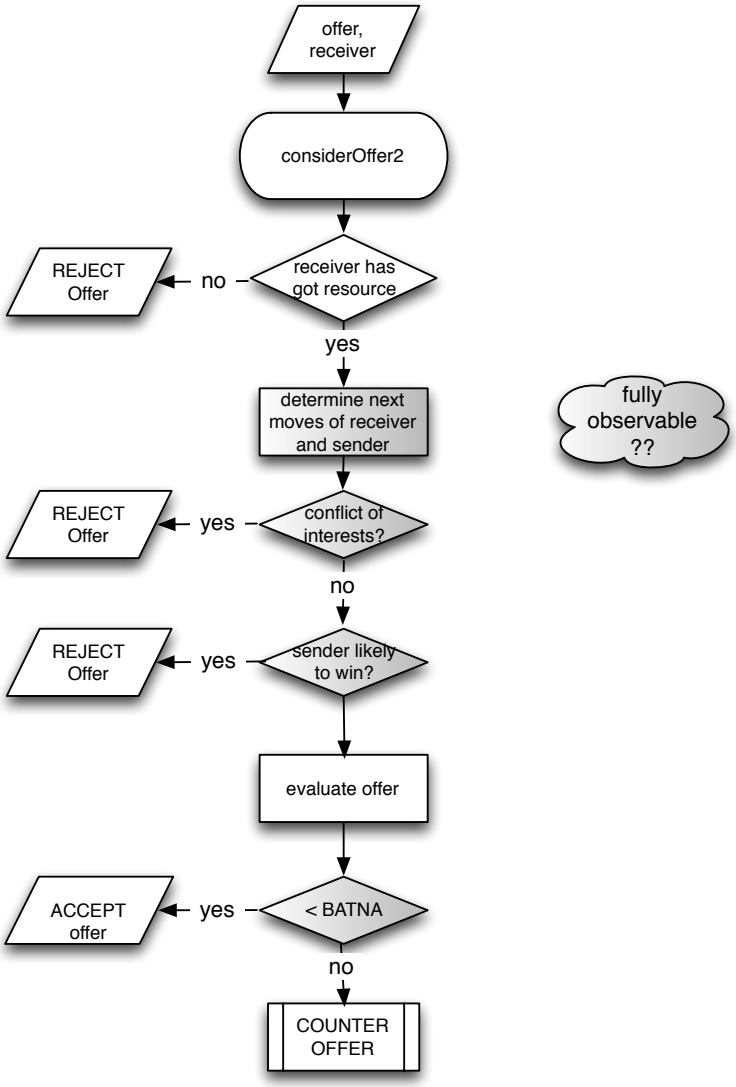

Figure 4: The `considerOffer` method from `SOCRobotNegotiator`.

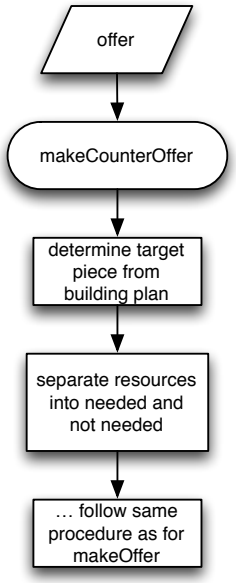

Figure 5: The `makeCounterOffer` method from `SOCRobotNegotiator`.

the possible trade offers, rank them and choose the best option. Currently, only the `makeOffer` method is implemented:

- Gather information
  - get target resources for BBP
  - get current resources
  - get potential trade partners (opponents that are not close to winning (VP < 8) and not in a race with me)
  - get legal (possible) offers to potential trade partners (considering our resources, consider: 1:1, 1:2, 2:1)
- Filter legal trades
  - trades that potential trade partners can do, based on our beliefs about their resources
- Rank trades
  - determine the trade that allows me to execute the BBP quickest
  - optional: require trades to be over a 'threshold of minimal improvement', i.e. reduce the estimated turns to win by the set minimal amount'
- Filter trades opponents are likely to do
  - optional step
  - uses an 'inverted considerOffer', i.e. I act as if I were the opponent, based on my beliefs about them (Note: This 'inverted considerOffer' still uses the original considerOffer method, so there may be 'incompatibilities' in

the way decisions are made.)

- Find the best trade
  - if there's such a trade, offer it
  - but only if we haven't offered it already (in this turn)

§ 9 DIALOGUE MANAGER

The `StacDialogueManager` works in conjunction with a robot's brain and is created in the constructor of `StacRobotBrain`. The dialogue takes place in the chat interface above the board by wrapping the messages in a `SOCGameTextMsg` message. When the server receives such a message it calls `handleGAMETXTMSG`, which parses the message for logging purposes and afterwards sends it to all the players participating in the game. When a client receives such a message from the server it either displays the message in the chat window if a human is playing or sends it to the robot brain for further processing.

The dialogue manager reacts to messages sent via the chat interface, when the brain's run loop calls the `handleChat` method. `handleChat` parses the message (based on the message structure, e.g. a build plan announcement has the form: `ANN:BP:p`, where `p` is the string representation of the `SOCPossiblePiece` obtained via `toString`) and decides what to do next. Message types are defined as static variables of this class and can be:

- Requests (for announcements of resources, build plans)
- Announcements (of participating in chat trading, resources or build plans)
- Trades (offer, accept, reject, force-accept)
- Comparisons (sent to the server for checks)
- Embargoes (request, comply or lift)
- Blocks (request, comply)

A robot player initiates the chat at the start of `PLAY1` state, by calling method `startTurnChat`, which will also update its current build plan. What it requests or announces is based on what type of robot it is. Decisions on embargoes are made by the `StacNegotiator`.

A robot's declarative memory keeps track of the players participating in trades by setting a boolean value in the `isParticipating` array for each player. Similarly, it keeps track of who is participating in the current negotiations by keeping String keys in the `isWaiting` array. These keys help the dialogue manager remember what it is waiting for from each player.

Human and robot players can negotiate both via the old trade interface or via the chat. A human player can decide how to trade as he/she will see the message in the chat interface and will also have the trade interface/speech bubbles pop up when receiving an offer or counteroffer.

As a precursor to proper use of natural language, a restricted natural language system is implemented in the static class StacChatTradeMsgParser (used by SOCPlayerClient) which translates the message written in the chat window to a TradeMessage that can be understood by StacDialogueManager and sends it to the server. At the moment this system can be used for sending full or partial offers/counteroffers and reject/accept offers received. Force-accepts, blocks or embargoes cannot be sent by a human player yet.

INTERACTION BETWEEN BRAIN AND DIALOGUE MANAGER. The dialogue manager doesn't execute all the logic behind the chat negotiations, rather it acts as a utility that is being called by the brain based on the logic in the run loop. Aside from the protected setters and getters, the brain interacts with the dialogue manager via four methods: `startGameChat`, `startTurnChat`, `handleChat` and `negotiateTrade`.

A robot will announce its participation in future chat negotiations just before placing the first settlement (in StacRobotBrain.`placeFirstSettlement`) via the `startGameChat` method. The current player initiates the chat by announcing information, requesting information or requesting trade offers from its opponents in the `startTurnChat` method, which will also update its current build plan. What it requests or announces, or if it even participates in the chat is decided based on what type of robot it is.

Decisions about embargoes are made in the StacNegotiator. `startTurnChat` is called at the start of the `PLAY1` phase of a turn, after rolling the dice.[3]

`handleChat` is called when the robot brain receives a `SOCGameTextMsg`. It parses the message and decides what to do next. See the list above for possible message types.

`negotiateTrade` is either used to respond to a new trade offer or it is called from inside the brain's `makeOffer` method; in both cases its purpose is to generate a new offer.

FINALISING A TRADE. The server keeps track of the current offer and all the responses to it, including any counteroffers. However, when the current player makes a new offer, the memory of the old negotiations is cleared so it is not possible to negotiate multiple offers at the same time.

When two players have agreed on a trade, the robot who proposed the offer/counteroffer that was accepted must send another *accept* message to the server accepting the player to trade with. In the case of a human player, a confirmation panel is displayed where the player must either accept or reject the negotiated offer. The server then processes the accepted offer

[3]Note: We might want to move this to the beginning of a turn, i.e. `PLAY` phase, if we want to negotiate something like: "I see you have a lot of development cards. If you are planning to play a knight before rolling, may I suggest targeting player X?".

SPECIAL CAPABILITIES. Issuing blocks or embargoes can only be done by robots with special capabilities (e.g. robots of type `NP_BLOCK_TRADES_NOT_MADE_TO_ME` or `NP_PROPOSE_EMBARGO_AFTER_OFFER_FOR_IMMEDIATE_BUILD`) and can also only be accepted by specific types of robots (e.g. `NP_BLOCK_TRADES_GULLIBLE` or `NP_COMPLY_WITH_EMBARGOES`). Blocks are only issued if (1) an offer was not made to the player or (2) the player's offer was rejected by everyone or (3) the player rejects a counteroffer. Embargoes can be proposed in the `startTurnChat` method or similar to a block. Another special ability is to declare a trade to be *force-accept*, which compels players susceptible to this move to accept the trade regardless of their own evaluation of its desirability. Again special robot types are required for this.

Finally, the current player may not want to initiate a trade, but instead ask its opponents to propose something. This is done in `startTurnChat` if the robot is of type `SOLICIT_TRADES`. When the opponents process the information sent by the current player, they may respond by making a new offer to the current player if they are of type `MAKE_SOLICITED_OFFERS`.

A CLOSER LOOK AT A FEW METHODS. Figures 1 to 3 contain pseudocode for the algorithms for the most convoluted methods in the dialogue manager.

HUMAN–ROBOT OR HUMAN–HUMAN CHAT NEGOTIATIONS. Chat negotiations and trades with non-robot players are performed in the same way as robot–robot negotiations, namely by negotiating on the resource sets to offer and receive via the chat interface while the actual trade is executed by sending a second accept message (as described in paragraph *Finalising a trade* on page 27). The only difference to the robot–robot negotiations is that a human player doesn't have to compose the whole `TradeMessage`, but can use a restricted formal language or simple natural language as described over the next paragraphs.

FORMAL TRADE LANGUAGE DEFINITION. Even though simple natural language can be handled by the system, an abstract language can be used for trading. In the grammar given below, R is the name of a resource, N the quantity of a resource, P a person, T the full trade message and B a bank (or port) trade.

- $N \in [1-99]$
- $R \in [clay, ore, sheep, wheat, wood \vee wd]$
- $P \in [0, 1, 2, 3, blue, red, white, orange, all]$
- $E \to [N]R$
- $E \to E, E$ (and)
- $T \to [Pl]; [Of]; [Re]$ (for partial offers, $[Of]$ or $[Re]$ can be omitted, but at least one needs to be present)

```
Data: the player number of the player that sent the message, the
      message received
Result: a list of responses to the trade message
update Negotiator ;
if we are a recipient then
    if we are waiting for a msg from the sender then
        set player response;
        stop waiting for a message from this player;
        if everyone has responded then
            handleNegotiations;
        end
    else
        if msg is a block or a No response then
            if we are not waiting for a trade response then
                do nothing ;
            else
                set player response;
            end
        else
            // this is a new offer to us
            set player response;
            handleNegotiations;
        end
    end
else
    // we are not included in the current negotiations
    if can block or embargo then
        send block or embargo or both;
    else
        send no response;
    end
end
```

Algorithm 1: `handleTradeMessage`

Data: current build plan
Result: a new offer
if *we have Jedi capability* **then**
  decide if it is worth using it;
  if *it is not productive to use it* **then**
    build offer filtering likely;
  else
    build offer without filtering;
    use JMT ;
  end
else
  build offer filtering likely;
end
if *we are capable of forcing trades* **then**
  try force trade;
end

Algorithm 2: `negotiateTrade`

- $B \rightarrow b[ank] : [Of]; [Re]$ (cannot be a partial trade offer)
- $Of \rightarrow o[ffer] : E$
- $Re \rightarrow r[eceive] : E$
- $Pl \rightarrow t[o] : P, P, P$ (*all* can only be used on its own)
- $Acc \rightarrow acc[ept] : P$ (P cannot be *all* and can only contain one player due to how robots parse trade messages)
- $Rej \rightarrow rej[ect] : P$ (P cannot be *all* and can only contain one player)

Players are identified by their colour or player number. Resources are specified either by writing the full name or a unique prefix (i.e. the first letter suffices with the exception of the distinction between *wheat* and *wood*. When making an offer to all other players, Pl can be replaced with *all*.

A trade is performed as follows:

- The player whose turn it is types: $t[rade] : Pl; o : E; r : E$ (but here the N in E is obligatory).
- The system sends the message to every player (whispering is not allowed).
- Only the players included in Pl can reply with either $acc[ept] : P$, $rej[ect] : P$ or $t[rade] : P; o : E; r : E$ (counteroffer), where P is the sender of the offer currently disputed. Robots will send a no-response (or acknowledgement) message when they are not recipients of the original offer.
- If somebody has accepted the offer, a window will pop up asking for the initiator to confirm the trade.
- If there are any counteroffers, the initiator may choose to accept one of them via the chat or continue the negotiations by sending a new offer.

```
Data:
Result: a list of responses
make list of responses;
if we have made an offer then
    if we are block gullible & we have received a block then
        reject all other offers;
        create a block comply message;
        return list;
    end
    if we are of type accept-better-counter then
        if there is a better counteroffer then
            create an accept message for it;
            return list;
        end
    end
    if we are of a type that can choose accepter & there is more than 1
    accept then
        choose accepter based on a metric;
        send offer only to the chosen accepter;
        set tradeMade flag to true;
    end
    if !tradeMade then
        handle accepts normally;
    end
    handle rejects;
    if !tradeMade then
        handle counteroffers as normal offer;
    end
    if !tradeMade & our turn then
        if we are capable of forcing trades then
            try force trade;
        end
    end
else
    handleOffer;
end
return list;
```

Algorithm 3: `handleNegotiations`

Example of a full offer:

- The current player (blue): $t[rade] : 1, white; o : 1wd, 1or; r : 1cl$ (I give 1 wood and one ore for 1 clay);
- Player 1 (red): $rej : 0$ (I reject your offer);
- White (2): $acc : 0$ (I accept your offer);
- Blue then accepts the trade with player 2 (white) via the interface.

Example of a partial offer:

- The current player (red): $t[rade] : all; r : 1clay$
- Player 0 (blue): $rej : 1$
- Player 2 (white): $rej : 1$
- Player 3 (orange): $t[rade] : red; o : 1clay; r : 1wheat, 1ore$ (I can give you 1 clay for 1 wheat and 1 ore)
- Player 1 (red): $acc : 3$
- Player 3 confirms via the interface

Counteroffers work similar to the partial trade example.

NATURAL LANGUAGE TRADING. The agents understand a range of English expressions, and the server will automatically detect if a trade has been agreed. Examples of possible expressions are:

- Peter, I give 1 wood & 1 clay for 1 sheep
- fr,s i give 2ore or whe for sheep &1wd (This translates to: "Fred, Sam, I give 2 ore or 1 wheat for 1 sheep and 1 wood.")
- I want ore
- I give 2
- Bank, I trade 4 sheep for 1 wheat. (This initiates a trade with the bank or a port.)
- No thanks, David.
- OK

Note that names of players, the bank/port and resources are matched against prefixes, e.g. 'ba' for 'bank', 'o' for 'ore' (also, 'wd' for 'wood'). Spacing and punctuation is optional to a great extent.

The recognised utterance structures for making an offer are:

. [playername], I have |I offer |I give |anyone want |anybody want [resource] for [resource]
. [playername], I want |I need |anybody got |anyone got |somebody got |someone got [resource] for [resource]

In the above, if the second resource is optional, this will be a partial offer. [playername] can be:

- Name of the recipient/robot as it appears in the hand panel
- Multiple players are separated by "," or whitespace
- Can be omitted (which means the offer is directed to everyone)
- For explicitly addressing all opponents, e.g. to override possible references to prior offers: all |anybody |everyone |everybody

[resource] can be:

- String describing up to 3 kinds of resources in the format [quantity type]
- Resources in conjunctive resource sets are separated by ",", "and", "&" or whitespace
- Resources in disjunctive resource sets are separated by "or"
- "or" is an exclusive-or ('I give wheat or wood' means 'I give wheat or wood but not both')
- Resource sets can either be conjunctive (default) or disjunctive but not a mixture of both
- Resources where no quantity is specified use quantity 1
- Resource quantities can be numbers or 'a |an |one |two |three |... |ten'
- Unique abbreviations for resource names can be used, e.g. 'c', 'cl', 'cla' for clay, etc., also: "wd" for wood
- If the resource type can be inferred, the quantity can be omitted ('I give 1'; 'I want 2')

Examples of offers are the first three items from the example list provided earlier.

Bank/port trades are also made via the chat interface as following:

- bank |port, I trade |I swap |I exchange [resource] for [resource]
- bank |port: [resource] for [resource]
- Example: Bank, I trade 4 sheep for 1 wheat.

This format is the same as for making offers, but the terms for recipient and both resources are compulsory. The first resource expression specifies the givables, the second one the receivables.

Rejecting and accepting offers is done as:

- No |No, thanks |No, sorry |Nope |Sorry |Maybe later [playername]
- OK |Okay |Yes |Yeah |Yep |Sure |Go on then |Yeah, sure |Alright [playername]

Where: [playername] is optional, depending on the context.

THE TRADE MESSAGE PARSER. `StacChatTradeMsgParser` is the utility that parses the formal language and simple natural language. The parser translates the chat message to a `TradeMessage` which can be understood by a robot. It is called when the player enters a string using the formal language before the client sends the chat message to the server. The correct format of offers (or counteroffers), accepts and rejects is defined using regular expressions. Before parsing and translating the message, `isTradeMsgFormat(String msg)` is called for matching the message to the known patterns.

In this translation step, feedback is provided in the player interface if a mistake is detected (such as: incorrect player number/colour, or offering and requesting the same resource type, or offering to yourself, or offering of unavailable resources etc).

If the translation succeeds, `composeTradeMessageString(TradeMessage msg, Integer toPn)` builds the correct `TradeMessage`. For a bank trade the process is similar by checking the message format `isBankTradeMsgFormat(String msg)`, then parsing the message which generates a `SOCBankTrade` object and sending this to the server. Again, feedback is provided in case something went wrong. In both cases, if the message doesn't match any of the known patterns, it is sent via the chat as it is, so the chat can still be used for its usual purpose.

§ 10 ESTIMATING BUILDING TIMES

`SOCBuildingSpeedEstimate` calculates approximately how long it would take a player to build something.

Method `calculateRollsFast` figures out how many rolls it would take the agent to get the target set of resources given a starting set. Parameters:

- `startingResources`: the starting resources
- `targetResources`: the target resources
- `cutoff`: throw an exception if the total speed is greater than this
- `ports`: a list of port flags
- return value: the number of rolls as a `SOCResSetBuildTimePair`

Algorithm:

- Set a cutoff, beyond which the number of rolls is not considered (set to 1000).
- Determine the possible trades with ports or the bank, depending on the player's available options.
- Determine which of the required resources takes the longest to acquire.
- Consider making the trade (with a port/the bank).
- Look into the future by iterating the above steps over the next rolls until

the agent acquires the desired resources.
- Do this, until the desired resources are acquired and return the number of rolls.

There is an analogous method `calculateRollsAccurate` that does the same as `calculateRollsFast` but gives a different (probably more accurate) estimate.

`recalculateRollsPerResource` updates global array `rollsPerResource` that contains the estimated number of rolls needed to acquire a resource from the board hexes, depending on the numbers and hexes that the agent is touching with his settlements and cities.

§ 11 MODELLING PLAYERS

`SOCPlayer` is the agent's model of a player.

In the `resources` array `SOCPlayer` stores the resources this player has.

Method `addRolledResources` adds to the player's resources (and resource-roll totals in `resourceStats`).

The array `resourceStats` is used by the server to keep track of how many resources (of each kind) the player got over the entire game by dice rolls.

Note that information about other players should now be mainly accessed via `StacDeclarativeMemory`. However, we distinguish between observable and unobservable information: unobservable information (beliefs) are all accessed via the `StacDeclarativeMemory`, whereas, observable information, e.g. a player's name, might still be accessed directly.

§ 12 THE SAVE AND LOAD FUNCTIONS

The utility that performs the save/load process is called `DeepCopy` in the *soc.util* package. As the name suggests it works by deep copying an object using Java's built-in serialisation process. The `DeepCopy` utility can also be used for duplicating objects without writing to a file (through the use of method `copy`).

SAVE. The data gathered when saving a game state is:

- the five `SOCGame` objects (one for each robot and one for the server),
- four `StacRobotDeclarativeMemory` objects if the robots are of type `Stac`,
- four `StacRobotBrainInfo` objects (containing the brain state for each of the robots playing),

- four `ArrayList` objects (containing the player trackers from each of the four robots' perspective).

Saving works by serializing this data and writing the bytes to corresponding files in a folder under the *saves* folder (see figure 6). This process can be initialised by a human player (by pressing the save button in hand panel of the player interface (figure 8) or by a robot. (Note that in the latter case the `saveGame` method in `SOCBrain` needs to be called depending on when saving is required). If a robot saves the game, then the files are stored in the *saves/robot* folder. When a human player saves the game, the files are stored inside a folder with the name formed of the game name and time stamp (e.g. *Practice_Mon_Mar_31_16_42_15_BST_2014*). This approach allows multiple saves by a human player, whereas a robot saving will overwrite the files contained in the *saves/robot* folder when performing multiple saves.

SAVING THE BOARD CONFIGURATION. The utility `SaveBoardConfigFromGame` reads the board configuration from a saved game and saves it inside a new file in the *saves/board* folder. Before running this utility, the saved `SOCGame` object belonging to the server (*server_soc.game.SOCGame*) should be moved to *saves/board* folder.

LOAD. Loading is the inverse process, done by reading the bytes from stored files and deserialising the bytes into new objects. It can be performed before starting a new game or any time during a game. In both case, the load is initialised by pressing the *Load* button (see figures 7 and 8) and then selecting the folder containing the saved files. Loading before starting a game allows to choose the position on the board, while loading during a game uses the existing configuration (e.g. if you are blue you will play as blue after loading).

Note that the actual loading is only performed after all players have sat down (regardless whether they are human or robot) and all the connections are established. As a result, the user must sit down on the spot he was occupying at the time of the save if he wants to continue the game as it was, otherwise he will play in the place of the player occupying the spot before.

The only data required for loading a game is the `SOCGame` object, so human players and robots can replace each other. However, a `StacRobotDeclarativeMemory` object declarative memory should be present if preferences/beliefs have to be preserved during the process of a type of robot replacing a different type of robot.

Both saving and loading are handled by the server and the four connected clients (`Server` and `SOCPlayerClient`/`SOCRobotClient`) through the normal message passing system (i.e. when the `processCommand` method encounters a `SOCGameCopy`, the game state is saved, while a `SOCLoadGame` initiates the load). Messages are being passed between server and clients via the

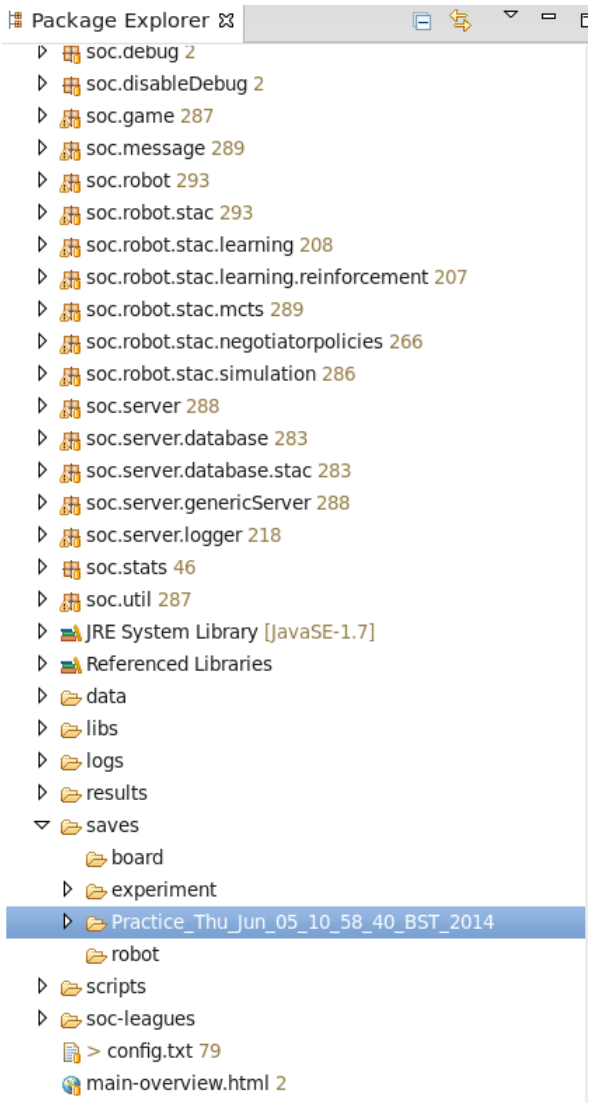

Figure 6: The folder tree.

Figure 7: The load button at the client start.

Figure 8: The load button on the player's hand.

normal route. When a robot is playing, the client suspends the robot's brain activity before executing any of the two processes. Due to the way `JSettlers` works, it also needs to update all the references between the replaced objects, e.g. both, `SOCGame` and `SOCPlayerTracker` need to have a reference to the same `SOCPlayer` data, while the `SOCPlayer` need to have a reference to the new `SOCGame` object; or `SOCPlayingPieces` needs to have their brain references updated etc. This piece of code is prone to errors as other parts of the project are being modified and it needs regular updates. (See the source code documentation for the `handleLoadGame`/`handleGameCopy` methods in the client classes for more information)

Note that the objects that need to undergo the serialization/deserialization process have to implement the `Serializable` interface. If any of the classes required for saving a game are modified, any previous saved information is lost, because Java will not be able to recognize the type of the stored objects. If the `SOCGame` is modified, any old saved games are completely lost.

§ 13 REPLAY UTILITIES

STAC POSTGRES DATABASE INTERFACE. Reading a full log file to extract the information from a played game is a slow process. For this reason, the game play (excluding the chat, but including the trade moves) from the corpus has been stored in a Postgres database. Each game is stored across three tables: one for the observable data, one for the extracted features and one for the actions/moves. The row IDs from the first two correspond to the states IDs and the IDs from the third table with the actions IDs as following: an action n results in the game state n formed of the information stored in the rows with $ID = n$.

`StacDBHelper` is the class that provides the necessary methods to connect to the database and interact with (read/write) the stored information. See source code documentation of the class for more information. New methods are added to this class as development progresses. `StacDBHelper` is contained in the *soc.server.database.stac* package alongside classes that describe the rows of the three tables of a game and a class that describe the row of a value function table. The url, user and password to access the database should be included in dbconfig.txt file in *resources* package.

REPLAY CLIENT. The `SOCReplayClient` makes it possible to replay games from log files in the *soclog* format. This class also creates an augmented log file, which prints out all the game information after each action executed in the game in addition to the normal game messages.

`SOCReplayClient` can also store the game states and actions from the logs as `ObsGameStateRow`, `ExtGameStateRow` and `GameActionRow`) inside a Postgres database, cf. the explanations above. Initially, it checks if it

can connect to the database and if the database contains the overall game information in tables games, players, seasons and leagues (this data must be added manually before running this class due to some consistency issues with the logs of the games). The collection process consists of two parts: one collects the overall number of trades and possible build plans executed by each player, the other collects the actual raw data and extracts new features (e.g. ETB, ETB, longest possible road for each player or other values that cannot be taken from the game state as it is and need further computing). For this reason, it needs to be executed twice in order to gather all information. The utility also provides the option to only extract new features, if the overall statistics and observable data have been collected already. These options can be selected by defining some command line arguments:

- if nothing is defined then it will just replay the game;
- `-c` to collect both observable and extracted data;
- `-c -eo` extract only, if the raw data has been already collected;
- any of the above with `-al` for creating an augmented log.

Note that the replay client collects the information based on a specific order of messages inside the logs. If this is modified in later versions, unexpected behaviour may occur.

STAC DB REPLAY CLIENT. Games that have been stored inside the Postgres database can be replayed using `StacDBReplayClient`. When launching, the tables containing a game's raw data and actions are looked up using the game's name.

REPRESENTATION: FEATURE VECTOR GENERATORS. The states and actions can be represented using vectors of either binary or numerical features (this is decided on in the `FVGeneratorFactory` which returns the correct generator). Offsets are defined in two interfaces for each of the two implementations for quick and easy access to the features. NOTE: When choosing binary or numerical features, make sure the classes accessing the vectors implement the correct offset interface. The code is in *representation* package.

§ 14 MONTE CARLO TREE SEARCH

MCTS. Monte Carlo Tree Search (MCTS) algorithm is an online sample-based method which chooses optimal moves by combining the exact nature of trees with random simulations. The algorithm is composed of four phases: selection, expansion, simulation and backpropagation (see figure 9).

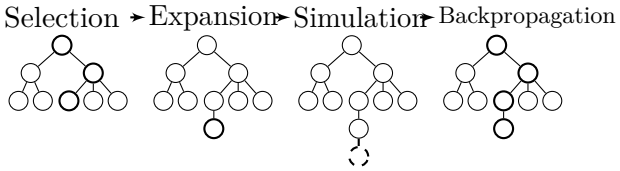

Figure 9: MCTS algorithm; Repeat the steps n times

MCTS EXTENDED TO THE FULL ACTION SPACE. An MCTS algorithm capable of handling the whole action space in the fully observable game was developed and is packaged in MCTS-1.0.jar. The jar was built from the code released alongside the paper (Dobre & Lascarides, 2017). `MCTSRobotBrain` is the interface of the MCTS algorithm to the `JSettlers` environment. `MCTSRobotType` contains a list of parameters that can be set beforehand.

SMARTSETTLERS. The smartsettlers implementation (Szita et al., 2010) can be found in the *originalsmartSettlers* package. A small set of fixes have been applied to make sure the simulation model follows the correct game rules (e.g. only 4 cities are allowed, the starting player is randomly picked before each game).

FLAT-UCT. Flat-UCT is a simplified version which only performs the expansion step for the root node. This implementation is in the *soc.robot.stac.flatmcts* package and can only be applied to the initial placement and robber actions of the game, due to the complexity of `JSettlers` code for deciding on the next legal action. For more information on how it combines the information from previous games with MCTS see (Dobre & Lascarides, 2015). The algorithm interacts with the `JSettlers` environment via `StacRobotBrainFlatMCTS` brain. As most of the actions of the game still use the heuristics for decision making, this class extends the `StacRobotFlatBrain` class. As a result, agent types are defined in the same way as for a normal Stac agent. The additional parameters are defined in `FlatMCTSType`.

§ 15 NOTES

`SOCTradeTree` is a tree that contains possible trade offers and how they're related to each other. Also contains a flag for weather or not this offer should be expanded to other offers.

`SOCDebug` and `SOCDisableDebug` contain the switch for printing debug output.

`SOCPlayerTracker` is used by the `SOCRobotBrain` to track possible building spots for itself and other players.

StacRobotDummyBrain was created to act as a container/utility provider for some methods that require access to information such as the game object, player trackers, player data and estimators.

StacRobotBrainRandom models the brain of a robot that can play a game randomly (while trades/negotiations are still made following the normal Stac policy). It contains two flags (`randomBuildPlan` and `randomInitial`) for random play or random initial placement. Another field has been added (`randomPercentage`, in the range $[-1, 100]$), which is used for deciding how often to play random. If the value is $n > 0$ then the robot plays $n\%$ of the time randomly, indifferent if it deciding during the normal game play or initial set up. Otherwise the decision is made based on the values of the first two flags (e.g. if `randomPercentage` $= 0$, `randomInitial` $=$ true and `randomBuildPlan` $=$ false, then the robot plays the initial set up phase completely random, while during the game it plays following a normal StacRobotBrain).

There are a set of commands for game actions that can be used by typing in the chat area instead of pressing the corresponding buttons on the interface:

- "\\roll" - roll dice
- "\\done" - end turn
- "\\buyr", "\\buys", "\\buyc", "\\buyd" - buy road/settlement/city/development card

References

Dobre, M. & Lascarides, A. (2015). Online learning and mining human play in complex games. In *Proceedings of the ieee conference on computational intelligence in games (cig).* Tainan, Taiwan.

Dobre, M. & Lascarides, A. (2017). Exploiting action categories in learning complex games. In *Ieee sai intelligent systems conference (intellisys).* London, UK.

Guhe, M. (2014). *Trading off Estimated Time to Build, Estimated Speedup and Estimated Progress in JSettlers* (Tech. Rep.). Edinburgh, Scotland: School of Informatics, Univeristy of Edinburgh.

Szita, I., Chaslot, G. & Spronck, P. (2010). Monte-carlo tree search in settlers of catan. In H. van den Herik & P. Spronck (Eds.), *Advances in computer games* (pp. 21–32). Springer.

Thomas, R. (2003). *Real-time decision making for adversarial environments using a plan-based heuristic* (Unpublished doctoral dissertation). Department of Computer Science, Northwestern University.