Implementation of a Poker autonomous and distributed system

Stefano Braggion

Facolta di Ingegneria e Scienze Informatiche, Universita degli Studi di Bologna
stefano.braggion2@studio.unibo.it

Abstract. Keywords: MAS · Jason · CArtAgO · TuCSon · ReSpecT.

Table of Contents

Implementation of a Poker autonomous and distributed system 1
Stefano Braggion
1 Texas Hold'em Poker 3
1.1 Cards 3
1.2 Suits 3
1.3 Hand values 4
2 Basic notions 4
2.1 Agent & Artifacts Paradigm 4
2.2 Jason and BDI 5
2.3 CArtAgO 5
3 System Architecture 8
3.1 Environment 8
3.2 Support packages 9
3.3 Card Model 9
3.4 Rank Evaluator 9
4 System Behavior 11
4.1 Game workflow 11
5 Action deliberation strategy 16
5.1 Preparation 16
5.2 Train 16
5.3 Decide 16
6 System Architecture 18
7 TuCSon Specifications 19
7.1 Join mechanism 19
7.2 Player's betting mechanism 20
7.3 State machine definition 22
7.4 Check mechanism 23
7.5 Retire mechanism 24
7.6 Round progress specification 26
8 Dashboard 27

Introduction

1 Texas Hold'em Poker

Texas hold 'em is a variation of the card game of poker. Two cards, known as hole cards, are dealt face down to each player, and then five community cards are dealt face up in three stages. The stages consist of a series of three cards ("the flop"), later an additional single card ("the turn"), and a final card ("the river"). Each player seeks the best five card poker hand from any combination of the seven cards of the five community cards and their two hole cards. Players have betting options to check, call, raise, or fold. Rounds of betting take place before the flop is dealt and after each subsequent deal. The player who has the best hand and has not folded by the end of all betting rounds wins all of the money bet for the hand, known as the pot.

1.1 Cards

A deck consist of 52 cards with 4 ranks and 14 cards for each rank.

Value	Card name
14	Ace
13	King
12	Queen
11	Jack
10	Ten
9	Nine
8	Eight
7	Seven
6	Six
5	Five
4	Four
3	Three
2	Two

1.2 Suits

Suit
Spades
Hearts
Clubs
Diamonds

1.3 Hand values

Here follows a description of hand values.

Value	Name	Description
1	Highcard	No combination of cards. Take the value of the highest card.
2	Pair	Two cards with the same value.
3	Two pairs	Two times two cards with the same value.
4	Three of a kind	Three cards with the same value.
5	Straight	Sequence of 5 cards in increasing value.
6	Flush	5 cards of the same suit.
7	Full house	Combination of three of a kind and a pair.
8	Four of a kind	Four cards of the same value.
9	Straight flush	Straight of the same suit.
10	Royal flush	Straight flush from Ten to Ace.

2 Basic notions

2.1 Agent & Artifacts Paradigm

The classic approach used for the implementation of environment is rooted in the classical conception of environment used in the field of Artificial Intelligence, a concept according to which the environment must only provide support to agents in terms of passive objects to be used. According to [1], in this perspective, the environment is therefore seen as a space in which agents can perform actions and retrieve information. However, the increasing complexity of multi-agent systems and therefore also of the characteristics of the environment, has favored the development of a new way of conceiving these spaces. This new point of view makes it possible to abandon the classical conception of environment, in favor of a new one which in fact considers the environment as a space in which the functionalities and support services to the agent are encapsulated. These types of environments are called endogenous systems and have a layer division based on the different functionalities.

- The first layer contains the basic mechanisms that allow agents to use the so-called deployment context, i.e. the external hardware and software resources made available (e.g. interface with sensors and actuators).
- The second layer, called abstraction level, allows to bridge the gap between the agent abstraction and the deployment context.
- Finally, the interaction-mediation level contains the mechanisms for concurrently accessing shared resources and for encouraging inter-agent communication.

The approach just described is based on the Agent&Artifact meta-model, which allows the designer to treat the environment as a real programmable space and conceive it in terms of artifacts and workspaces. The term artifact

indicates a dynamic object with a computational capacity. Finally, artifacts are organized within workplaces. However, the definition of artifact as a dynamic object must not be completely compared to the definition of a classic object in object-oriented programming.

2.2 Jason and BDI

The agent paradigm bases its foundations on the philosophical implications on human reasoning affirmed by the famous Michael Bratman. One of the main consequences of this is the fact that an agent can be conceived as having a mental state. The model on which AgentSpeak is based derives from these implications and is called *BDI - Belief, Desire, Intention.*

- *Beliefs* are the information that the agent has on the environment in which it is located. They can derive from their own perceptions of the environment or from communication with other agents.
- *Desires* are instead possible objectives that an agent could bring to completion. This does not mean that the agent will definitely carry them out, they are just different options.
- Finally, *Intentions* are desires for which the agent begins a series of actions with the aim of reaching the goal. An agent can pursue goals because they have been delegated by other agents, or following a choice.

Jason is an implementation of AgentSpeak, a specific language for the development of multiagent systems based on the BDI paradigm just described.

```
/* Initial beliefs */
        started.

/* Initial Goals */

/* Plans */

+started <- .print ("Hello World").
```

Listing 1.1. Jason sample Hello-World

2.3 CArtAgO

CArtAgO is the acronym of *Common ARTifact infrastructure for Agents Open environment* and it is based on the Agents & Artifacts meta-model, providing support for the development and execution of environment.

The environment consists of a dynamic set of computational entities called artifacts, which represent resources and tools that agents can use.

Figure 2 shows instead the abstract representation of an artifact. An artifact exposes a set of observable properties and operations available to agents. The observable properties are based on the observer pattern and are variables that

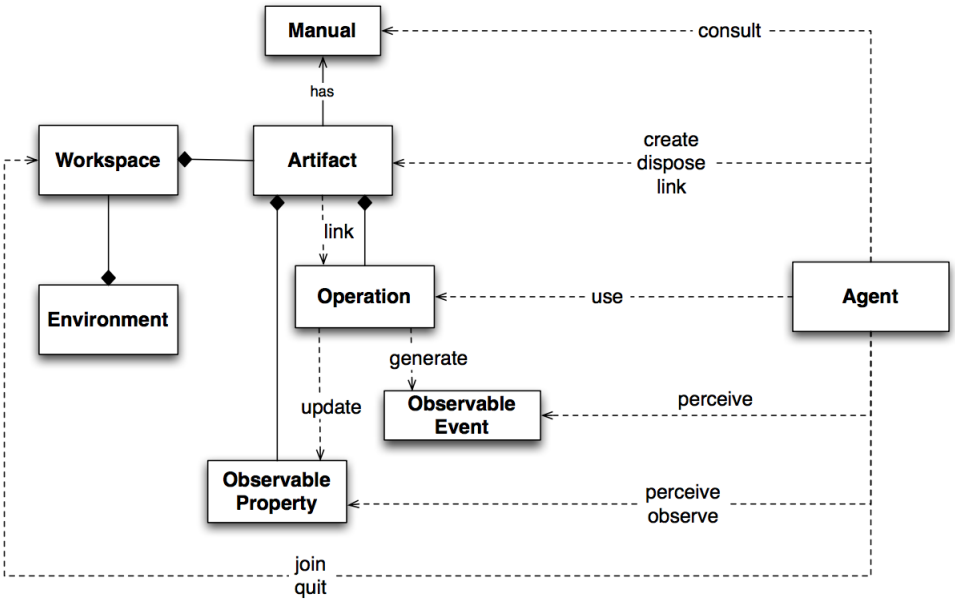

Fig. 1. CArtAgO meta-model

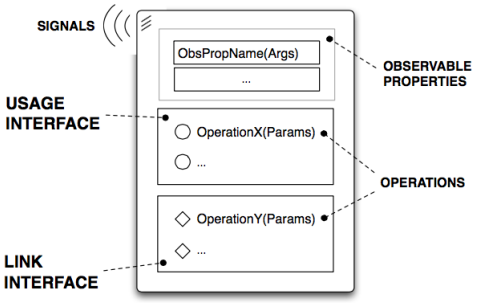

Fig. 2. Abstract model of an artifact

can be observed by the agents. The operations are instead the functionalities, in the form of processes, executed within the artifacts. The execution of an operation may or may not generate signals, which are used to convey certain information.

System project and implementation

3 System Architecture

3.1 Environment

The environment in which agents are situated is composed by several artifacts which acts as proactive computation entities and coordination medium. The environment is divided into two packages: *evaluator* and *model*.
In the below list you can find a list of all artifacts used (Fig. 4, Fig. 3):

- **DecisionArtifact**. It is the "brain" of the player. Each time it has to deliberate an action, *call* or *fold*, the agent create an artifact of this kind and uses the Monte Carlo methodology to try to guess the possible combinations.
- **EvaluableHandArtifact**.
- **EvaluatorArtifact**.
- **DeckArtifact**. Represents a deck composed by 52 cards.
- **PotArtifact**. Represents the pot. Players and Dealer use the pot as a form of synchronization and to place money.
- **TableArtifact**. Represents the table in which the dealer put on the uncovered cards.
- **StatusArtifact**. Represent the game status, own the currently active players and manage the state (current and next) of the game.

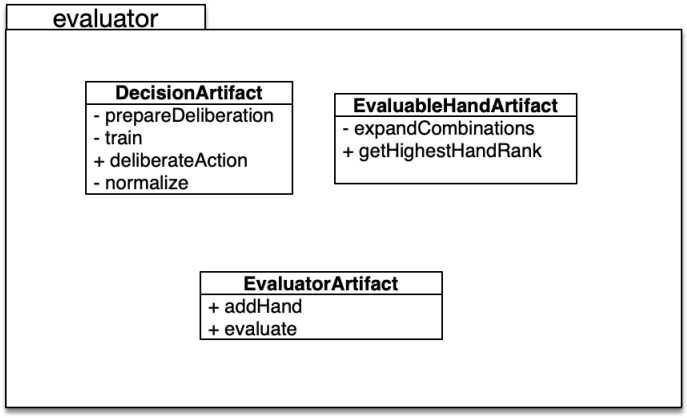

Fig. 3. Composition of evaluator artifact package

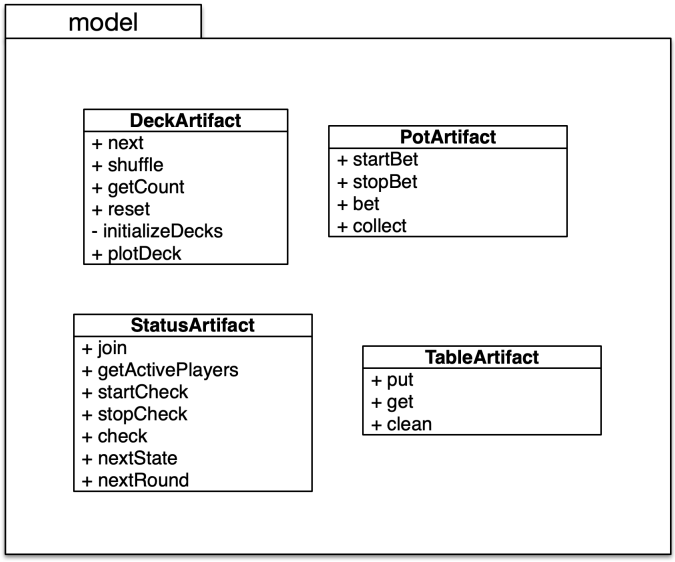

Fig. 4. Composition of model artifact package

3.2 Support packages

Not all objects needs to be represented as Artifacts. At this purpose, a series of *POJO* (Plain Old Java Object) classes have been defined in order to support artifacts management.

3.3 Card Model

This package contains all the classes used to model a Card. A Card is composed by a suit (CLUBS, DIAMONDS, HEARTS, SPADES) and a value (TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE).

- **Card**. Represent a card.
- **CardSuit**. Represent a suit for a card.
- **CardValue**. Represent a value for a card.
- **CardList**. Class used to represent a hand (set) of cards.
- **Rank**. Enumeration that models all the possible combination cards result.

3.4 Rank Evaluator

This package contains all the concrete evaluator, one for each kind of possible combination (Fig. 6).

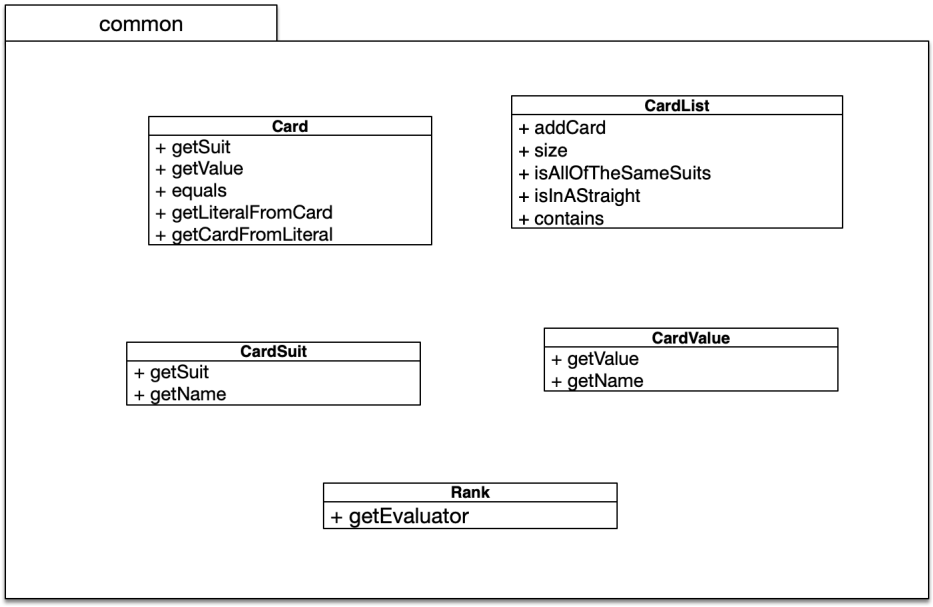

Fig. 5. Card model package

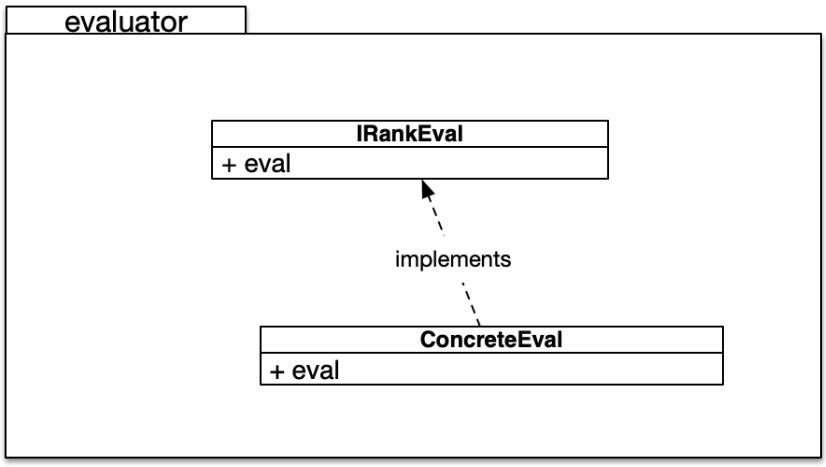

Fig. 6. Rank evaluator package

4 System Behavior

The overall system behavior is composed by multiple states, sensed by the players and the dealer, in order to execute the relative plan and act.

State	Player Behavior	Dealer Behavior
init	Wait for dealer signal	Initialize the system.
blind_bet	Big blind and small blind bet	Start the bid.
pre_flop	Receive 2 cards.	Send 2 covered cards to each player.
pre_flop_bet	Bet a fixed amount of 2 cash.	Start the bid.
flop	Wait for next state.	Put 1 uncovered card on table
flop_bet	Bet a fixed amount of 2 cash.	Start the bid.
turn	Wait for next state.	Put 1 uncovered card on table
turn_bet	Bet a fixed amount of 2 cash.	Start the bid.
river	Wait for next state.	Put 1 uncovered card on table
river_bet	Bet a fixed amount of 2 cash.	Start the bid.
showdown	Send the best combination to dealer	

4.1 Game workflow

The game is composed by several rounds in which every player can *decide* to continue playing or to wait for next. The game begins with the dealer who initializes game assets (artifacts): the table, the pot, the deck of cards and the status, after that sends to all players the *ready* belief (See fig. 7). Players can now join the game. They interact with the *status* artifact in order to do so. In the next phase, represented in fig. 8, the dealer tell to *small blind* and *big blind* to place their bet. In this context, the *pot* artifacts is not only an entity that models a game asset, but has to be considered a real coordination tool. Subsequently, the game status enter in the *pre-flop* state (Fig. 9). In this state the dealer sends 2 covered cards (hole cards) to each player and tell them to place their fixed amount bet (Fig. 10). Next state is the *flop*, in which the dealer places 3 uncovered cards on the table, and tell each player to deliberate an action (See section 5). Now the game status goes through 2 equal phases (*turn*, *river* - Fig. 11) interspersed with 2 betting phases (12). In these 2 states, the dealer places an uncovered card on the table, and tell each player to deliberate an action. Now we can have two main different results:

- Only 1 player remain active in current round. In this case, the dealer stops all active operations and decrees the winner.
- Two or more players remain active. The dealer goes to the *showdown* state.

During the showdown stage (Fig. 13, each player gets the highest combination of cards (hole cards and table cards) and sends it to the dealer, who will evaluate all the combination and decide the winner.

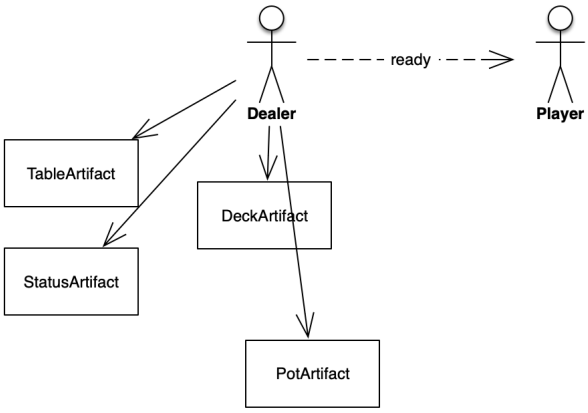

Fig. 7. Init state

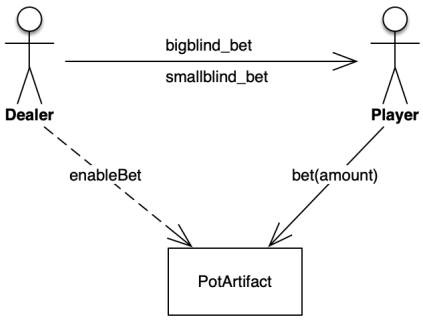

Fig. 8. Blind_Bet state

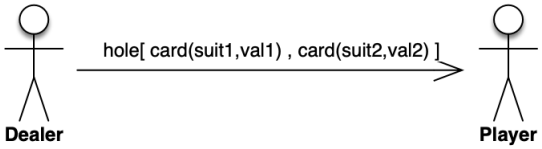

Fig. 9. Preflop state

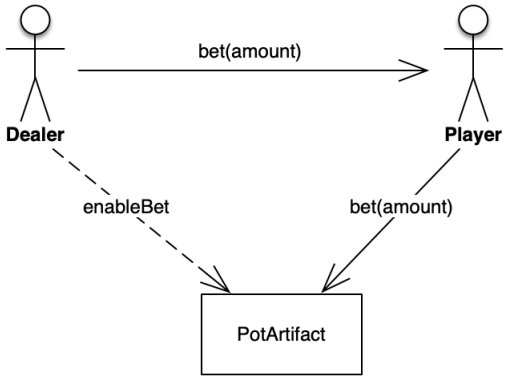

Fig. 10. Preflop_Bet state

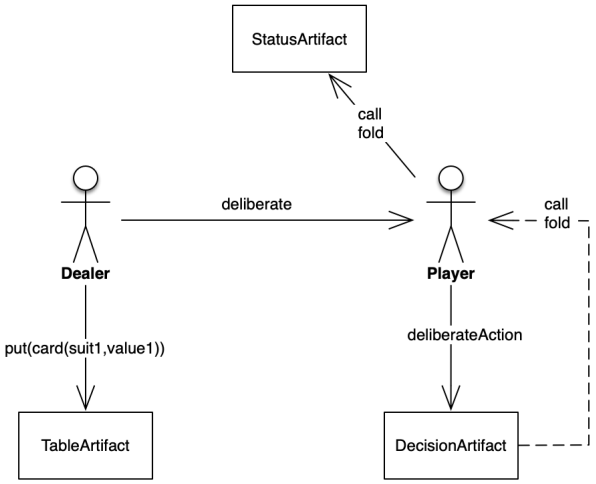

Fig. 11. Flop/Turn/River state

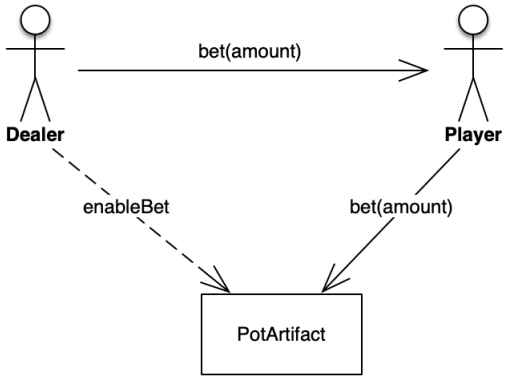

Fig. 12. Flop/Turn/River Bet state

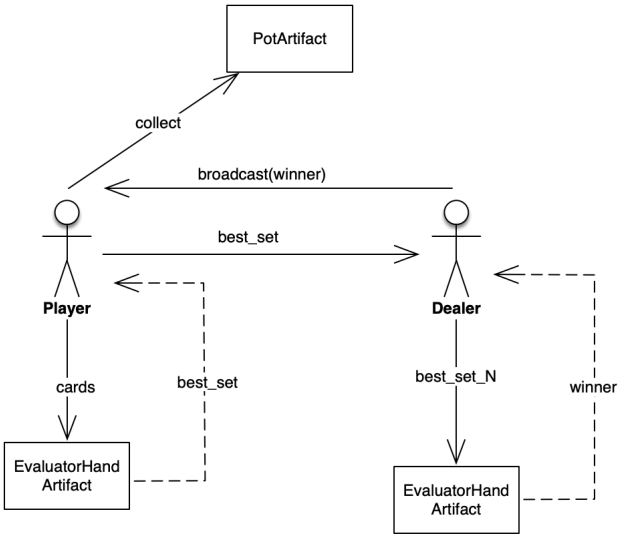

Fig. 13. Showdown state

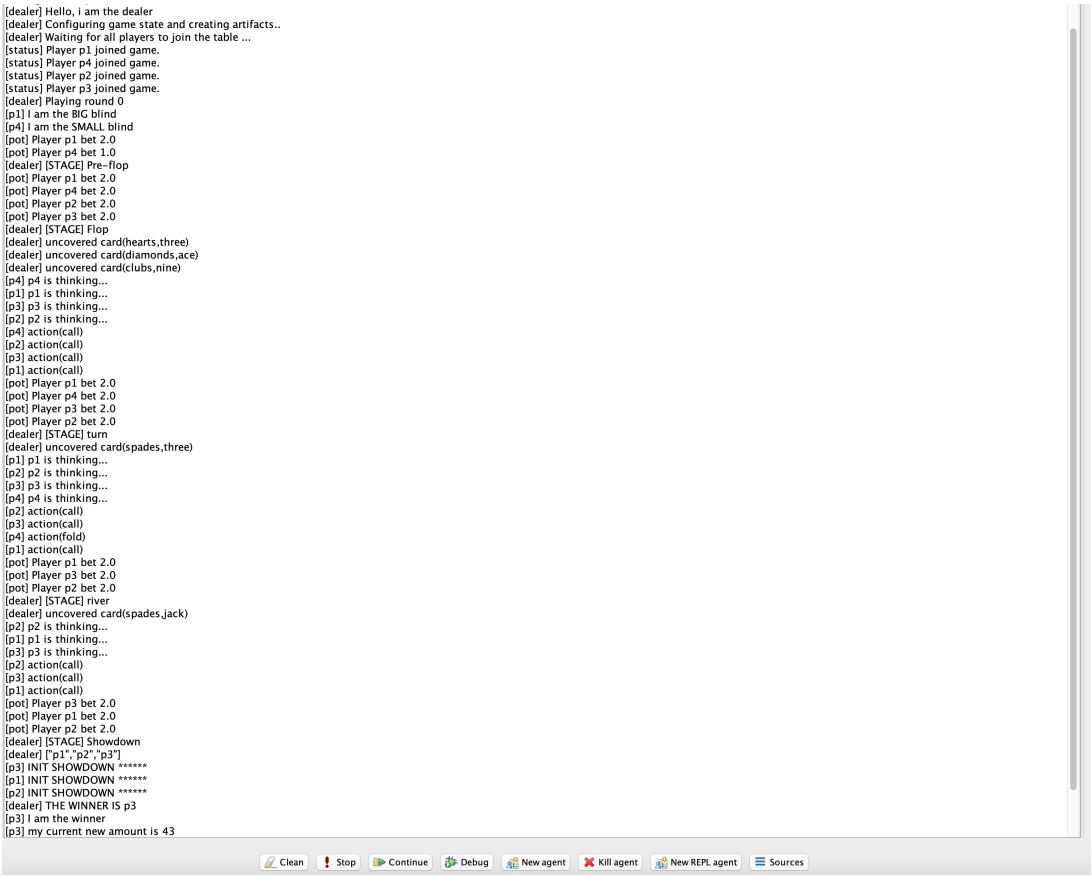

Fig. 14. Sample output

5 Action deliberation strategy

The core of the player mind is represented by the decision about what to do. Every time the player has to do some considerations about the action to take (*call* to stay in game, *fold* to leave out for that round), an *decision* artifact is created to support the reasoning. In this way, it is as if one of the thinking elements of the agent were outsourced, possibly located anywhere in the net.

As we can see in listing 1.3, the deliberation of an action is divided into several sub-operations.

5.1 Preparation

This operation transform the cards (in the form of Prolog literals) to a set of Java objects that can be used for computation.

5.2 Train

Purpose of this stage, is to literally train the player making him play a large number of hands, so to have an accurate estimate of the probability of the single combinations (see for example list 1.2).
Let's say, for example, game status is in the flop stage. In that context, each player have the 2 hole cards and there are 3 uncovered cards on the table. A player has 5 cards and try to guess which cards will come out, gradually evaluating the various combinations. The first step is the creation of a virtual deck cleaned from the cards that he currently knows, so to avoid duplicates. Decision artifact then creates all possible hand , in order to expand all combination of 5 cards with 7 given.

```
Four of a Kind: 0 events
Full House: 101 events
Royal Flush: 0 events
Straight: 0 events
Three of a Kind: 505 events
One Pair: 13534 events
Flush: 0 events
High Card: 6565 events
Two Pairs: 1515 events
Straight Flush: 0 events
```

Listing 1.2. Sample train odds result

5.3 Decide

In the last stage of deliberation, the player make some reasoning about the calculated odds. In that stage, it is considered also the "personality" of the player, represented by a literal in the belief base. The personality represents the degree of risk that the player is willing to have. All hands rank have an associated

value ranging from 1 (High Card) to 10 (Royal Flush). Let's say for example that a player consider *good* all hands rank from Flush (5) to Royal Flush (10). The decision method calculate the upper and lower sums of the probabilities deriving from the train phase. If the difference between the upper sum and the lower sum is positive, then the result action is *CALL*, otherwise *FOLD*.

```
@OPERATION
void deliberateAction( Object[] hCards,
    Object[] tCards, OpFeedbackParam<Literal> result) {

        execInternalOp("prepareDeliberation", hCards, tCards
            );
        await("trainGuard");

        execInternalOp("train");
        await("decideGuard");

        execInternalOp("decide");
        await("resultGuard");

        result.set(this.resultAction);
}
```

Listing 1.3. Deliberate action operations

Towards a distributed model of the system

The system described so far, is totally located within a single node. Now we want to literally spread the system parts over the network. In this perspective, we can have different players on different nodes, the dealer on another one and so on. At this purpose, TuCSon is introduced as the coordination media between the agents.

TuCSoN (Tuple Centres Spread over the Network) is a general-purpose agent-oriented model and infrastructure for MAS coordination. TuCSoN is based on a coordination model providing tuple centres (tuple space enhanced with the possibility to program its be- haviour in response to interactions) as first-class abstractions to design and develop general-purpose coordination artifacts. TuCSoN tuple centres are programmed through the ReSpecT logic-based specification language [9].

6 System Architecture

As we can see in Fig. 15, now all agents are enriched with a personal tuple centre in which they can store and retrieve tuple and also trigger operation. From the seat tuple centre, all operations are directed to the shared tuple centres (status and table). So, in this perspective, all the communication protocol is embedded into the tuple centres.

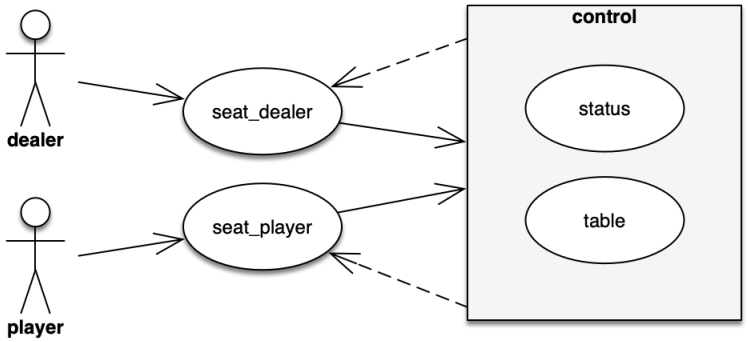

Fig. 15. Agents and TuCSon integration

The introduction of the Seat tuple centre allows the player to have to know only this object to communicate with, completely decoupling the logic of the status from the player. Each time wants to perform an operation or communicate a decision, a player emits a specific tuple in his SeatTC. In this implementation, coordination tuples can be of two types:

- *op(opname(param))* for the operations.
- *action(actionname)* for the actions.

7 TuCSon Specifications

7.1 Join mechanism

When a player wants to join the game, he can emit a tuple in his personal seat. The seat performs a remote operation on the *StatusTC* which give permission to play. In that perspective, a player ask the system to join the game. If the maximum number of players has not yet been reached, then the system update the current number of players and the state.

```
reaction(
   out(op(join(Name))),
   (link_in, invocation),
   (
      rd(current_players(Cur)),
      rd(max_players(Max)),
      Cur < Max,
      Cur2 is Cur + 1,
      in(current_players(Cur)),
      out(current_players(Cur2)),
      out(player(Name))
   )
).

reaction(
   out(op(join(Name))),
   (link_in, completion),
   (
      in(op(join(Name)))
   )
).
```

Listing 1.4. Join mechanism specification (StatusTC)

```
%% player join operation
reaction(
   out(op(join(Name))),
   (operation, invocation),
   (
      status@localhost:20504 ? out(op(join(Name)))
```

```
    )
).

reaction(
    out(op(join(Name))),
    (link_out, completion),
    (
        in(op(join(Name)))
    )
).
```

Listing 1.5. Join mechanism specification (SeatTC)

7.2 Player's betting mechanism

In this game implementation, there are some state in which it is necessary for players to make some bet and put money in the pot. These phases are completely managed by the TC infrastructure. Each time is reached a state that involves a bet phase, a tuple to start that phase is emitted by the *StatusTC* (Code 1.8). A player emits a tuple in his seat each time he makes a bet. The tuple is of the type *op(bid(A))* (Code 1.6). As we can see in the follow reaction specification, the *SeatTC* trigger a remote operation on the *TableTC*. In Code 1.7 the table reacts to the tuple and increases the value of the pot. Finally, the table perform a remote operation on the status in order to manage how many players have already bet.

```
reaction(
    out(op(bid(A))),
    (operation, invocation),
    (
        table@localhost:20504 ? out(op(dobid(A)))
    )
).

reaction(
    out(op(bid(A))),
    (operation, completion),
    (
        rd(money(M)),
        M1 is M - A,
        in(money(M)),
        out(money(M1)),
        in(op(bid(A)))
    )
).
```

Listing 1.6. Betting mechanism specification (SeatTC)

```
reaction(
   out(op(dobid(A))),
   (link_in, completion),
   (
      rd(current_pot(A1)),
      A2 is A1 + A,
      in(current_pot(A1)),
      out(current_pot(A2)),
      in(op(dobid(A))),
      status@localhost:20504 ? out(op(bid(done)))
   )
).
```

Listing 1.7. Betting mechanism specification (TableTC)

```
%% start bet
reaction(
   out(op(start_bid)),
   internal,
   (
      rd(current_players(Cur)),
      rd(current_state(State)),
      State == blind_bet,
      out(nplayersbid(2))
   )
).

reaction(
   out(op(start_bid)),
   internal,
   (
      rd(current_players(Cur)),
      rd(current_state(State)),
      State \== blind_bet,
      out(nplayersbid(Cur))
   )
).

%% player bet done
reaction(
   out(op(bid(done))),
   (link_in, invocation),
   (
      rd(nplayersbid(Cur)),
      Cur > 0,
      Curnew is Cur - 1,
      in(nplayersbid(Cur)),
      out(nplayersbid(Curnew))
   )
).
```

```
reaction(
   out(op(bid(done))),
   (link_in, completion),
   (
      in(op(bid(done)))
   )
).
```

Listing 1.8. Betting mechanism specification (StatusTC)

7.3 State machine definition

Below are the specifications of the reactions used to model the state machine for the poker game. Each state is represented by a tuple. In the invocation phase of the operation, we check the current state and them modify it accordingly to the system rules. In the completion phase, simply delete the tuple that triggered the operation.

```
%% river_bet -> showdown
reaction(
   out(op(next_state)), (invocation, operation),
   (
      rd(current_state(S)),
      S == river_bet,
      in(current_state(S)),
          out(current_state(showdown))
   )
).

%% river -> river_bet
reaction(
   out(op(next_state)), (invocation, operation),
   (
      rd(current_state(S)),
      S == river,
      in(current_state(S)),
      out(current_state(river_bet))
   )
).

%
%      ........    other states
%


reaction(
   out(op(next_state)), (operation, completion),
```

```
    (
        in(op(next_state))
    )
).
```

Listing 1.9. Definition of state machine

7.4 Check mechanism

Every time the dealer must change state, must wait for each player to make his action. To accomplish this, a check mechanism was introduced. As can be seen from the underlying specifications, each player emits a tuple *action(check)* in the personal *SeatTC*. Once all player have checked, the dealer can change status.

```
%% check
reaction(
    out(required(check)),
    internal,
    (
        rd(current_players(Cur)),
        out(check_players(Cur))
    )
).

%%player check
reaction(
    out(action(check)),
    (invocation, link_in),
    (
        rd(check_players(Cur)),
        Cur > 0,
        Cur2 is Cur - 1,
        in(check_players(Cur)),
        out(check_players(Cur2))
    )
).

reaction(
    out(action(check)),
    (invocation, link_in),
    (
        rd(check_players(Cur)),
        Cur == 0,
        in(required(check))
    )
).

reaction(
    out(action(check)),
    (completion, link_in),
```

```
    (
        in(action(check))
    )
).
```

Listing 1.10. Check mechanism specification (StatusTC)

```
reaction(
    out(action(retire(Name))),
    (operation, invocation),
    (
        status@localhost:20504 ? out(action(retire(Name)))
    )
).

reaction(
    out(action(retire(Name))),
    (link_out, completion),
    (
        in(money(M)),
        in(action(retire(Name)))
    )
).

reaction(
    out(action(check)),
    (invocation, operation),
    (
        status@localhost:20504 ? out(action(check))
    )
).

reaction(
    out(action(check)),
    (link_out, completion),
    (
        in(action(check))
    )
).
```

Listing 1.11. Check mechanism specification (SeatTC)

7.5 Retire mechanism

Another possible action for a player is to completely retire from game. Every time a new round starts, a player must check if have enough money to play the entire round. If not, the player must retire from game. At this purpose, a set of reactions have been implemented in the tuple centres.

```
reaction(
   out(action(retire(Name))),
   (operation, invocation),
   (
      status@localhost:20504 ? out(action(retire(Name)))
   )
).

reaction(
   out(action(retire(Name))),
   (link_out, completion),
   (
      in(action(retire(Name))),
      in(money(M))
   )
).
```

Listing 1.12. Retire mechanism (SeatTC)

```
reaction(
   out(action(retire(Name))),
   (link_in, invocation),
   (
      rd(check_players(Cur)),
      Cur > 0,
      Cur2 is Cur - 1,
      in(check_players(Cur)),
      out(check_players(Cur2)),
      rd(current_players(P)),
      P2 is P - 1,
      in(current_players(P)),
      out(current_players(P2)),
      in(player(Name,enabled))
   )
).

reaction(
   out(action(retire(Name))),
   (link_in, invocation),
   (
      rd(check_players(Cur)),
      Cur == 0,
      in(required(check))
   )
).

reaction(
   out(action(retire(Name))),
   (link_in, completion),
   (
```

```
        in(action(retire(Name)))
    )
).
```

Listing 1.13. Retire mechanism (StatusTC)

7.6 Round progress specification

The game is organized into several rounds. When a round ends, some cleaning operations must be performed.

- Enable all players in the StatusTC.
- Reset the small blind and big blind
- Update the number of current players in the game
- Remove all cards from the table
- Remove the money from the pot and reset it to zero
- Remove the cards from the SeatTC

```
reaction(
    out(op(next_round)),
    (operation, invocation),
    (
        rd(current_round(R)),
        R1 is R + 1,
        in(current_round(R)),
        out(current_round(R1)),
        rd(small_blind(Sb)),
        in(small_blind(Sb)),
        rd(big_blind(Bb)),
        in(big_blind(Bb)),
        rd(current_state(St)),
        in(current_state(St)),
        out(current_state(init))
    )
).

reaction(
    out(op(next_round)),
    (operation, completion),
    (
        in(op(next_round))
    )
).
```

Listing 1.14. Next round (StatusTC)

```
reaction(
    out(op(next_round)),
    (operation, invocation),
```

```
    (
        rd(current_pot(P)),
        in(current_pot(P)),
        out(current_pot(0))
    )
).

reaction(
    out(op(next_round)),
    (operation, completion),
    (
        in(op(next_round))
    )
).
```

Listing 1.15. Next round (TableTC)

8 Dashboard

In order to better understand the evolution of the game, a simple dashboard was introduced.

- In the status panel (Fig. 17) is reported the current situation of the game (current state, current amount of money in the pot, who's the small blind and big blind, etc..).
- The players panel (Fig. 18) contains messages and actions for players.
- The Table panel (Fig. 19) contains messages and actions from dealer. It contain also the uncovered cards on the table.
- The common log panel (Fig. 20) contains all other kind of messages from all system components.

List of Figures

1 CArtAgO meta-model 6
2 Abstract model of an artifact 6
3 Composition of evaluator artifact package 8
4 Composition of model artifact package 9
5 Card model package 10
6 Rank evaluator package 10
7 Init state 12
8 Blind_Bet state 12
9 Preflop state 12
10 Preflop_Bet state 13
11 Flop/Turn/River state 13
12 Flop/Turn/River Bet state 14

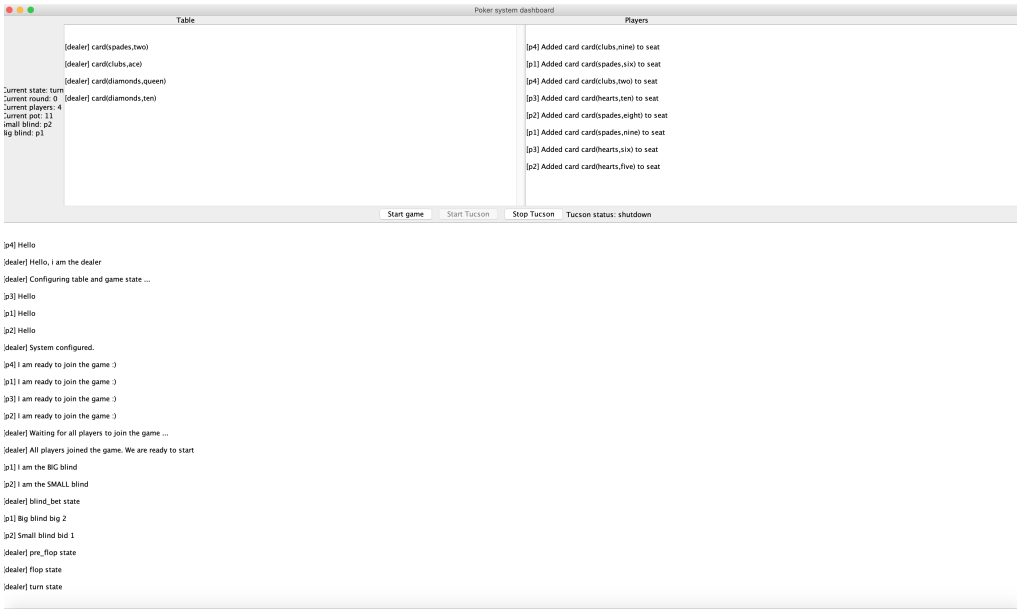

Fig. 16. GUI dashboard

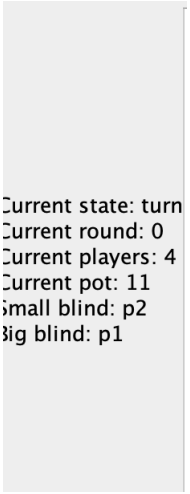

Fig. 17. Status panel

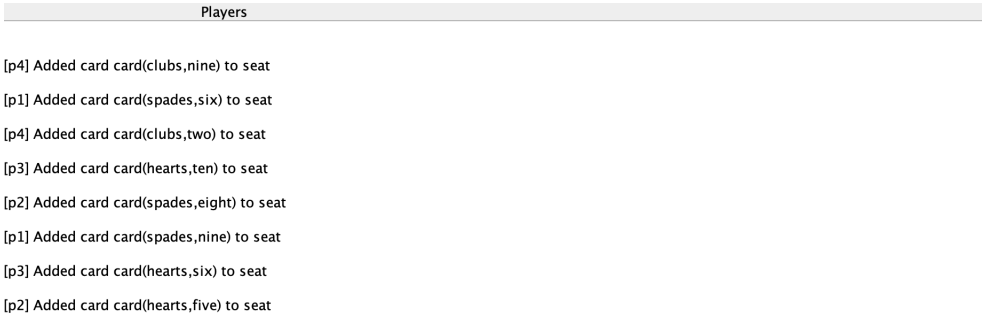

Fig. 18. Players panel

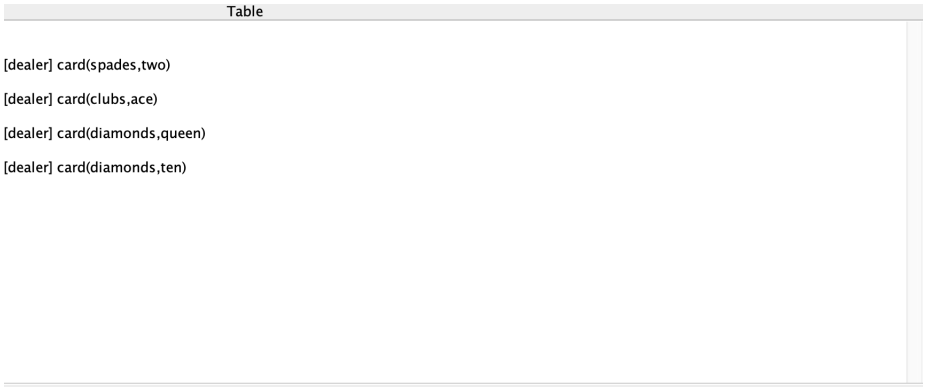

Fig. 19. Table panel

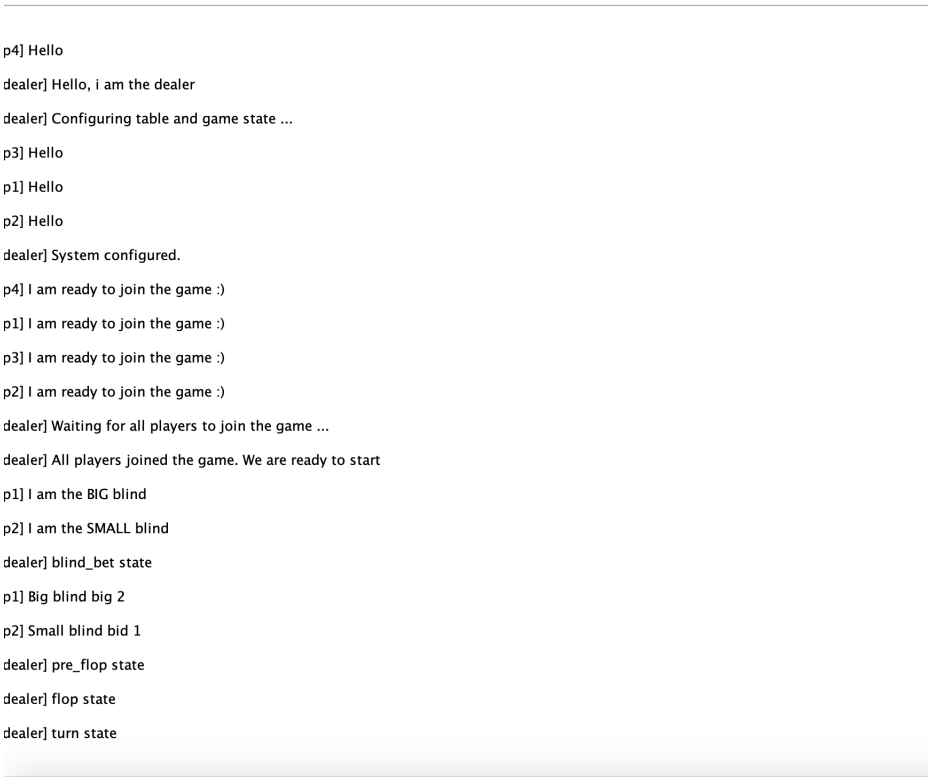

Fig. 20. Common log panel

13 Showdown state 14
14 Sample output 15
15 Agents and TuCSon integration 18
16 GUI dashboard 28
17 Status panel 28
18 Players panel 29
19 Table panel 29
20 Common log panel 30

References

1. Braggion S. :
Programmazione di Sistemi Multi-Agente: la piattaforma Jason come caso di studio.
Facoltá di Ingegneria Cesena, Universitá degli studi di Bologna, 2015.
2. Omicini A. :
Material from Autonomous Systems course
Universitá degli studi di Bologna
3. Omicini A. :
Material from Distributed Systems course
Universitá degli studi di Bologna
4. Bordini R., Hubner J. , Wooldridge M. :
Programming multi-agent system in AgentSpeak using Jason.
Wiley, 2007
5. Padgham L. , Winikoff M. :
Developing Intelligent Agent Systems: A practical guide.
Wiley, 2004
6. Ricci A., Piunti M. , Viroli M. :
Environment programming in multi-agent systems: an artifact based perspective
Springer US, 2010
7. Jason website documentation.
http://jason.sourceforge.net
8. CArtAgO website documentation
http://cartago.sourceforge.net
9. Omicini A. :
Formal ReSpecT in the A&A Perspective
Electronic Notes in Theoretical Computer Science 175 (2007)
10. TuCSoN website
http://apice.unibo.it/xwiki/bin/view/TuCSoN/WebHome