A Concise Tutorial on GAVS+

March 15, 2013

Chih-Hong Cheng

Department of Informatics, TU München, Germany
Fortiss - An-Institut der TU München,, Germany
Email:{cheng.chihhong@gmail.com}

http://www6.in.tum.de/~chengch/gavs

CONTENTS

I. Tool description and license

GAVS+ is an open-source tool which enables to visualize a broad spectrum of algorithmic games used in verification and synthesis, and offers a standard interface with utility functions to establish connection with engineering practice. It is developed by Department of Informatics (Unit 6), Technische Universität München, and is served for research and educational purposes. The software is released under the GNU General Public License (v3).

Libraries which GAVS+ used explicitly include

- JGraph: Java package for the manipulation of graphs.
- JDD: BDD package for symbolic manipulation of sets of Boolean variables.
- SAT4J: SAT solver based on Java.
- Apache Common Math 2.0: light-weight mathematic and statistic library for Java.
- PDDL4J: Front-end parser library for PDDL.

The software (executable .jar) and its source code are fully available at http://www6.in.tum.de/~chengch/gavs.

II. Thirty seconds for a quick taste of GAVS+

[Open the main window] After downloading the package, extract the compressed file. Double-click the **`dist`** folder. Double-click **`GAVS+.jar`** to invoke the main window. Note that executing GAVS+.jar requires libraries located in the **`dist/lib`** folder.

A. Reachability game

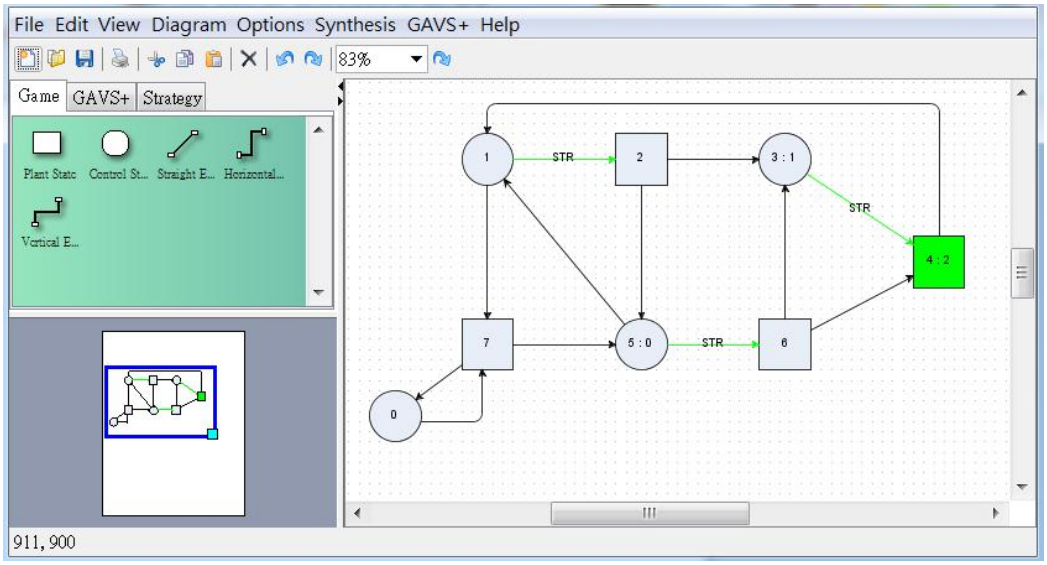

Figure 1. A reachability game and the generated strategy to win the game.

1) Open the file **`TestCase1_Reachability.mxe`** in the "**`GAVS_TestCase`**" folder. The specification and the strategy are already shown.
2) On the menu bar, choose **`Synthesis -> Clear Strategy Label`** to remove the highlighted strategy.
3) Choose **`Synthesis -> Reachability Game -> Graphical Spec`** to generate the strategy based on the graphical specification.
4) The strategy is generated by the back-end engine, and indicated on the game arena again.
5) When observing the result, we find that vertex 0 does not have strategies to reach vertex "4:2", while others can. The strategy marks an edge with green label **`STR`**.

B. Simple stochastic game

For a simple stochastic game, it tries to answer whether it is possible to reach vertex **`P0SINK`** with probability > 0.5.

1) Open the file **`Sample.mxe`** in the **`GAVS_plus_TestCase/SSG`** folder.
2) On the menu bar, choose **`GAVS+ -> Simple Stochastic Game -> Strategy Improvement (Hoffman-Karp)`**.
3) The strategy will be generated by the back-end engine, and indicated on the game arena.
4) On the **`Strategy`** panel, the probability for each vertex to reach `P0SINK` is listed.

III. Visualizing games and generating strategies: two-player, turn-based, finite arena

In this section, we consider two-player, turn-based games with various winning conditions, including reachability, safety, Büchi, parity, weak-parity, Staiger-Wagner, Muller, and Streett. Contents of this section covers the scope of our ATVA paper [3].

A. Construct the game graph

1) Once when GAVS+ is opened, an empty arena is available for the user to draw the game graph. On the "Game" panel, the user can use drag-and-drops to create vertices and edges in the arena. Some notices:

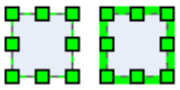

Figure 2. Vertex resizing (left) and edge creating (right).

- GAVS+ supports vertex resizing. Please refer to Figure 2: when a node is

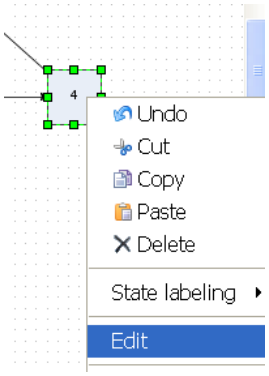

Figure 3. Right click on a vertex and edit its name.

- Make sure that all vertices have unique names (otherwise the engine will report an error). This is done by right-clicking the selected state, and choosing the **`edit`** option, similar to Figure 3.

2) The user may store his immediate results of the graph by selecting **`File -> Save`** operations.

B. Create the specification

GAVS+ supports both graphical and textural methods for the creation of specifications.

- For safety games, the user can label risk states with **`red markers`**; after invoking the synthesis engine, the strategy will try to avoid reaching these red-labeled states.

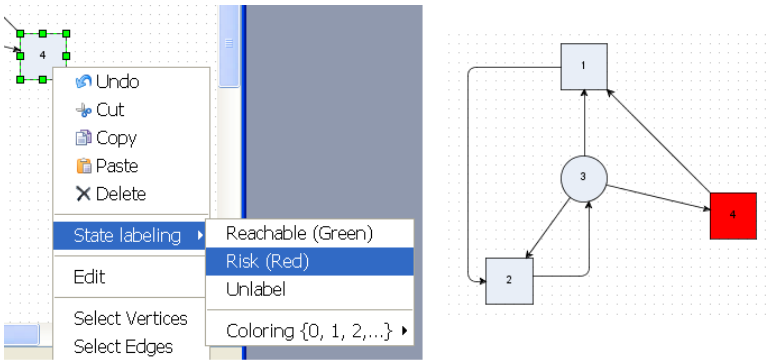

Figure 4. Label a state to be "risk" (left) and the resulting graph (right).

  - To do this, right click on the targeted risk vertices, and choose "State labeling' Risk (Red)", similar to Figure 4.
- For reachability/Büchi games, follow the same method as safety games, and the state will be marked with **`green`** color.

- For weak-parity or parity games, every state must be labeled with colors (which are non-negative numbers). Right click on the selected vertices, and choose **`State labeling -> Coloring`** {**`0, 1, 2...`**}. An example for the resulting parity game can be found in Figure 5.
  - For colors greater than 8, the user can directly edit the state with the format **`VertexName : VertexColor`** to achieve the same result.

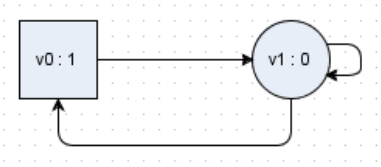

Figure 5. A simple parity game with its vertex coloring.

- Textural specifications can be edited using normal text-editors. It is suggested to store the specification file ended with ".**`spe`**".
  1) For Staiger-Wagner games, each row of the specification indicates one set of states $F_i \subseteq PowerSet(V_0 \uplus V_1)$.
  2) For Muller games, each row indicates the set of vertices which should be visited infinitely often.
  3) For Streett games, each row should be of the following format **`Ei : Fi`**, meaning that the E set and the F set are separated by "**`:`**".

C. Execute back-end synthesis engines from GAVS+

Invocation of synthesis engines for turn-based games in GAVS+ are called from the menu bar **`Synthesis`** (Figure 6).

Figure 6. The synthesis menu bar in GAVS+

- For games with positional strategies, the generated strategies will be shown automatically on the graph with edges labeled with "**`STR`**", similar to Figure 1.
- For safety games, all allowable (safe) transitions will be highlighted.
- For Staiger-Wagner games, the FSM strategy will be shown on the **`Strategy`** panel.
  - **[EXAMPLE]** We show an example where the goal is to reach all vertices $\{v_0, v_1, v_2\}$ at least once for the Staiger-Wagner game in Figure 7.
    1) Open the file **`TestCase5_StaigerWagner.mxe`** in the **`GAVS_TestCase`** folder, and on the menu bar, choose **`Synthesis -> Staiger-Wagner Game`**.
    2) Select the specification file **`TestCase5_StaigerWagner.spe`**. Then press the texttStrategy panel. The strategy will be shown.
    3) In the `Strategy` panel, the memory content ordering of variables will be shown; in this example, the ordering is $[v_2, v_1, v_0]$.

D. Execute back-end game reduction engines from GAVS+

For Muller and Streett games, instead of generating strategies directly, game reductions are implemented for clearer understanding regarding the meaning of strategies. This is due to the fact that the generated FSM strategies for Muller and Streett games require memory which is in the worst case factorial to the number of states.

- After the reduction action is invoked, the original Muller/Streett game graph and the newly generated parity graph will **COEXIST** in the canvas (for the use of comparison). If you want to invoke the parity solver, **BE SURE TO REMOVE THE OLD GRAPH**.

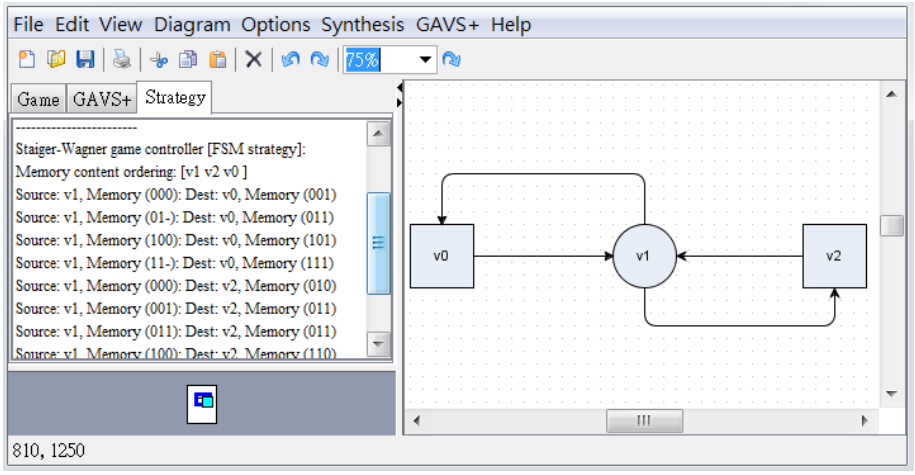

Figure 7. A simple Staiger-Wagner game.

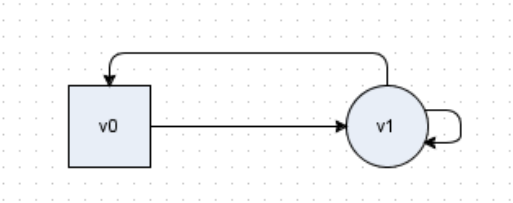

Figure 8. A simple game with Muller condition $\{\{v_0, v_1\}\}$.

Example We show an example where the goal is to reach all vertices $\{v_0, v_1\}$ infinitely often in Figure 8. There are some more complicated examples, but here for simplicity reasons we use this one.

1) Open the file **`TestCase6_SimpleMuller.mxe`** in the **`GAVS_TestCase`** folder, and choose **`Synthesis -> Game Reductions -> Muller to Parity`**.
2) Select the specification file **`TestCase6_SimpleMuller.spe`**, which is also in the folder.
3) The resulting graph can be rearranged using the automatic layout function; in this case, use **`Diagram -> Layout -> Vertical Horizontal`**, and the resulting graph is similar to Figure 9.

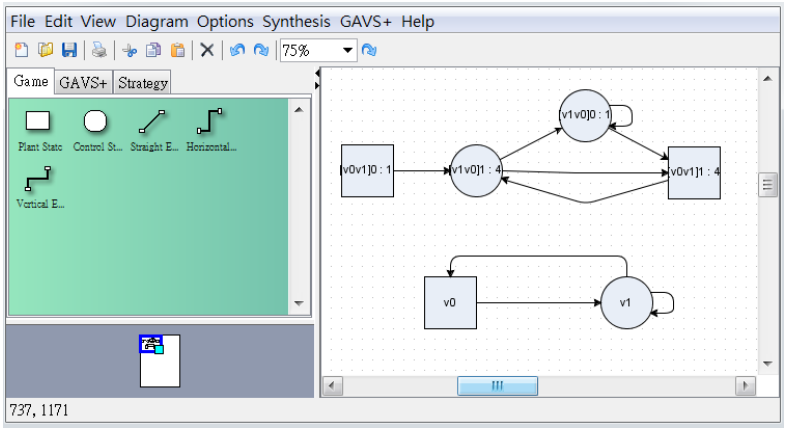

Figure 9. A simple game with Muller condition $\{\{v_0, v_1\}\}$.

4) Remove the original graph, and invoke the parity solver via **`Synthesis -> Parity`**. The result informs that at vertex v_1, player 0 should visit vertex v_0.

Formata for the vertex of the generated parity graph are as follows:

- For Muller games, it is of form **`[vertex permutation] LAR : Color`**. LAR stands for *latest appearance record*, indicating the index from which the first element in the permutation is retrieved. Details are omitted here.
- For Streett games, it is of the form vertex name **`[index permutation][Last index of E][Last index of F]: Color`**. It is based on the *index appearance record (IAR)*. Details are omitted here.

E. Observe intermediate results of synthesis engines from GAVS+

For educational purposes, GAVS+ also features visualization of intermediate steps during the synthesis process. This operation is triggered by the option **`Visualization of Intermediate Results`**. Currently this is restricted to some algorithms which performs symbolic executions (reachability, safety, Büchi, weak-parity).

An example is shown in Figure where a reachability game is played: starting from the desired goal state v_4, the 0-attractor region is increased gradually. As for the result, the set of all green states the last figure constitutes the winning region W_0 of the game.

Remark:

1) For the Büchi game, the starting point of simulation will be the set of states representing the recurrence region (a subset of the original targeted set F).
   - *Example* Open the file `TestCase4_BuechiGraphical.mxe` in the `GAVS_TestCase` folder. Then execute **`Synthesis -> Buechi Game -> Visualization of Intermediate Steps`**.

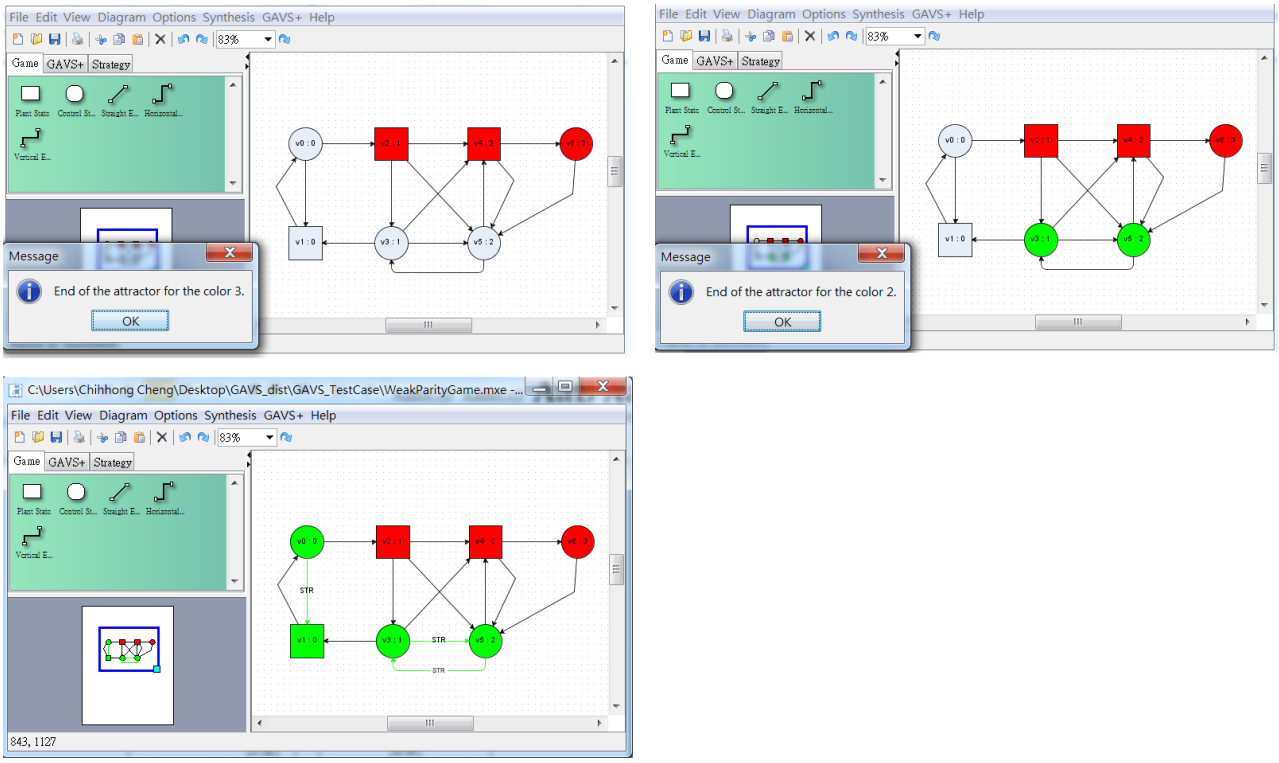

Figure 10. Intermediate steps for executing weak parity games.

2) For the weak-parity game, each iteration will indicate the set of states constituting the attractor starting from the highest color vertices. An example can be found in **`TestCase10_WeakParity_Graphical.mxe`**, where Figure 10 illustrates the intermediate steps: from highest color 3 to 0, GAVS+ uses two colors to partition winning regions of W_0 and W_1.

IV. Extended game support in GAVS+

In this section, we discuss the extended support of game types in GAVS+. These games are in general of strong practical use, and are currently under active research. Table 1 summarizes the games supported by GAVS+. To execute the engine of these game types, it is always under the menu **GAVS+**.

Table I
GAME TYPES AND IMPLEMENTED ALGORITHMS IN GAVS+, WHERE "†" INDICATES THAT THE GAME TYPE OR THE ALGORITHM IS IMPLEMENTED IN THE PREVIOUS VERSION [3], "‡" INDICATES THAT NO VISUALIZATION IS AVAILABLE, AND "^{u}" IS UNDER DEVELOPMENT.

Game type (visualization)	Implemented algorithms
Fundamental game†	Symbolic†: (co-)reachability, Büchi, weak-parity, Staiger-Wagner Explicit state†: parity (global/local discrete strategy improvement) Reduction†: Muller, Streett
Concurrent game	Sure reachability, almost-sure reachability, limit-sure reachability
Pushdown game‡	reachability (positional min-rank strategy, PDS strategy), Büchi (positional min-rank strategy, PDS strategy), parity reduction
Distributed game	Reachability (bounded distributed positional strategy for player-0), safety based on antichains^u
Markov decision process	Policy iteration, value iteration, linear programming (LP)
Simple stochastic game	Shapley, Hoffman-Karp, Randomized Hoffman-Karp
Games of incomplete information	Algorithms based on lattice theory and antichains

A. Extensions for pushdown systems

For games with infinite states, our interest is in games played over push-down graphs (APDS), a natural extension when recursion is considered. Currently no visualization of APDS is possible. We will use recursive games for the visualization in our future version.

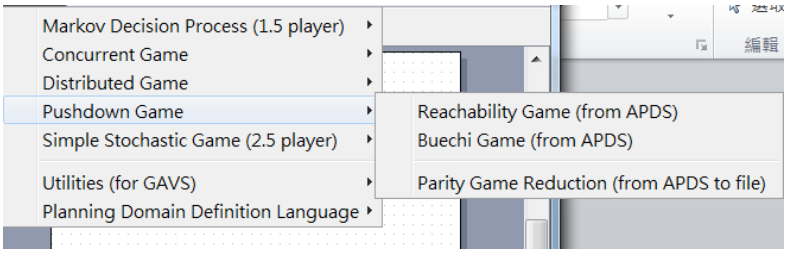

Figure 11. The menu bar for solving APDS.

- To solve the pushdown game for the reachability criteria, execute **GAVS+ -> Pushdown Game -> Reachability Game (from APDS)**, similar to Figure 11.
- Once when the strategy is found, GAVS+ asks the user which strategy he/she wants to execute for interactive mode. The supported strategy includes:
  1) Positional min-rank strategy and
  2) PDS strategy.
- For Büchi games, positional min-rank strategy simulation is offered.
  - In this version, we follow the algorithm of T. Cachat in [2], meaning that for games satisfying the Büchi condition, currently the engine solves a restricted form (although it is equivalent to the general form).For details, please see the paper.
  - Under this restriction, the goal configuration should be of the type $\{P\}$, meaning the set of all configurations which has states starting with location P, i.e., $P \cdot \Sigma^*$.

Example Consider the example in **GAVS_plus_testcase/APDS/example1.pds**, where it describes an APDS with the format the same as Figure 12.

The screenshot of interactive execution (including all choices available by player-1) is shown in Figure 13.

- Result: In this example, the interactive simulation starts from selecting the interactive simulation type. Select **Positional Min-rank Strategy**.
- The user is now playing the role of player-1 (spoiler) and selects the move based on his wish.
- Once when GAVS+ receives the move from the player, it performs the update based on the Positional min-rank strategy. For the above example, GAVS+ pops out the window as an indication of the next move.

```
## Comments used in the pds file (example1.pds)
P0_STATE = {P0}
P1_STATE = {P1}
ALPHABET = {a}
RULE = {P0 a -> P0; P0 a -> P0 a a; P1 a -> P0; P1 a -> P0 a}
INIT = {P1 a a}
GOAL = {P0 a a}
```

Figure 12. A simple APDS (example1.pds)

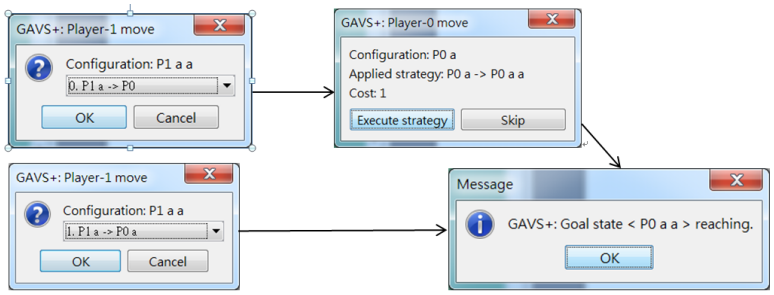

Figure 13. The tree of complete interactive simulation for APDS in Figure 12.

B. *Extensions for probabilistic systems*

In this branch, we consider cases when uncertainty is introduced in the game. Markov decision process (MDP) [11] is an optimization model on decision making in a stochastic environment, which is widely used in economics and machine learning. It consists of controllable and probabilistic vertices, which are called states and actions, respectively. Stochastic games [10] are games consisting of controllable, uncontrollable, and probabilistic vertices. In GAVS+, we focus on simple stochastic games (SSG) [4].

1) Visualization and synthesis of MDP: To visualize MDP, it is required to create actions (stochastic vertices), which can be found in the "**GAVS+**" panel (diamond shape, see Figure 14).

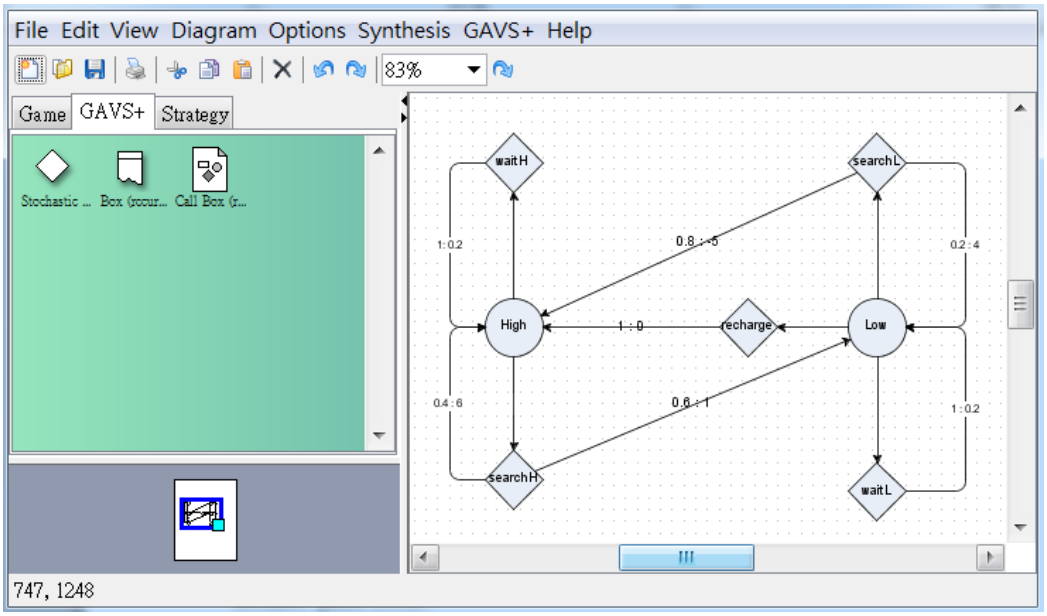

Figure 14. Constructing MDPs using the diamond vertices.

The example (**GAVS_plus_Testcase/MDP/MDP.mxe**) above is adapted from http://www.cse.lehigh.edu/~munoz/CSE335/, which models a robot having two battery levels `High` and `Low`, and actions for `waiting`, `searching`, and battery `recharging`. For a given action, it may change the battery level with certain probability, and generates some reward. For example, in the action `searchH`, the label "`0.4:6`" means that the action may go to location High with probability 0.4, and generate reward of value 6.

- The goal is to generate a strategy which optimizes the reward. Note that a discount value between the interval $[0, 1)$ is specified to avoid generating infinite reward.
- Once when the game graph is constructed, to generate the strategy with intermediate steps, execute **`GAVS+ -> Markov Decision Process -> Infinite Horizon Discounted -> Policy Iteration`**.
- Visualization of intermediate steps is also possible in GAVS+.
- To clear the strategy label, please select **`GAVS+ -> MDP -> Clear strategy labels (for MDP)`**. This is used to ensure that the only the ***action label*** is cleared.

Currently in GAVS+, we have implemented three algorithms for solving MDP:

1) Value iteration (with visualization of intermediate steps)
   - **[Notice]** For value iteration, the algorithm stops when the calculated value is very close to the previous calculated value (difference ¡ 0.0001); as the source code is fully available, users can modify this value freely.
2) Policy iteration (with visualization of intermediate steps)
3) Linear programming (using the `SimplexSolver` in Apache Common Math library).

2) Visualization and synthesis of SSG: For SSG, similar construction techniques can be applied similar to the construction of MDP. The only remark is that to label P0Sink (similarly P1Sink), users simply annotate the vertex with texts "**`:P0SINK`**", similar to the labeling of color in two-player, turn-based games when solving parity games.

Figure 15 shows the example **`GAVS_plus_Testcase/SSG/Sample.mxe`**, where the optimal response for control and plant is highlighted.

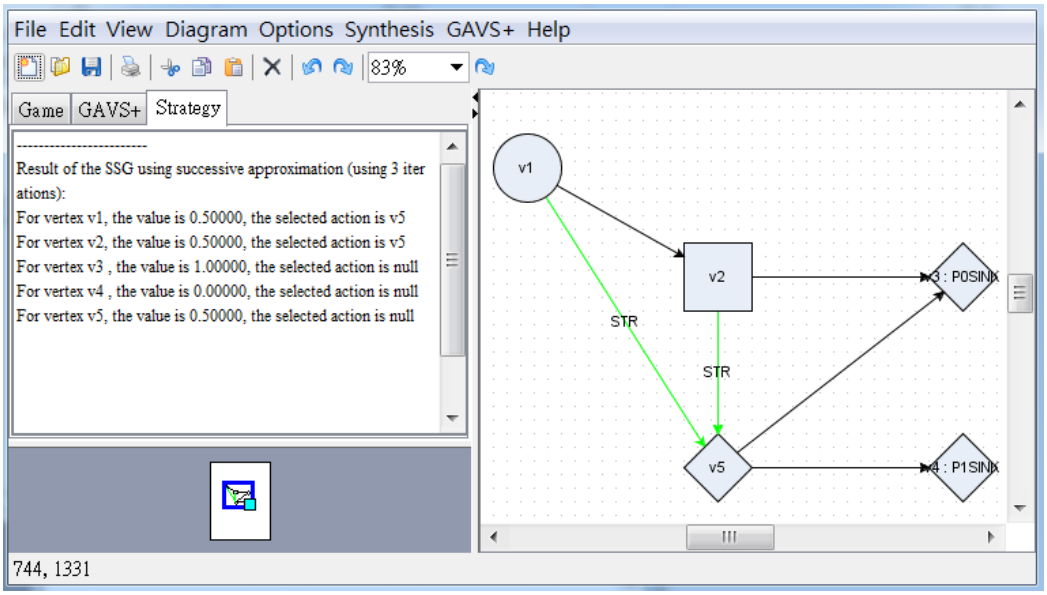

Figure 15. An SSG with labeled strategies.

C. Extensions for concurrent systems

Concurrent games are used to capture the condition where the control and environment simultaneously select their moves, and the next location is based on the combined decision. The engine in GAVS+ is able to solve the following winning conditions in a concurrent game:

1) Sure reachability winning
2) Almost-sure reachability winning (see example `/GAVS_plus_TestCase/LeftOrRight.mxe`)
3) Limit-sure reachability winning (see example `/GAVS_plus_TestCase/HideOrRun.mxe`)

Example Open the file **`LeftOrRightGame.mxe`** in the **`GAVS_plus_TestCase/CRG`** folder. This example is reused from [5], where player-0 continuously throws a snow ball on the left or right window, and player-1shows up each time on either the left or the right window.

- For this game, player-0 can hit player-1 (reaches state `S_hit`) with probability 1 from `S_throw`. This is called *almost-sure winning*.
- To generate the strategy, execute **`GAVS+ -> Concurrent Game -> Almost-sure Reachability Winning`**.
- On the strategy panel, the engine prints out the almost-sure winning region. We observe that for `S_throw`, both actions `throwL` and `throwR` are listed. This means that player-0 should play a random strategy which executes `throwL` with probability 0.5 and executes `throwR` with probability 0.5.

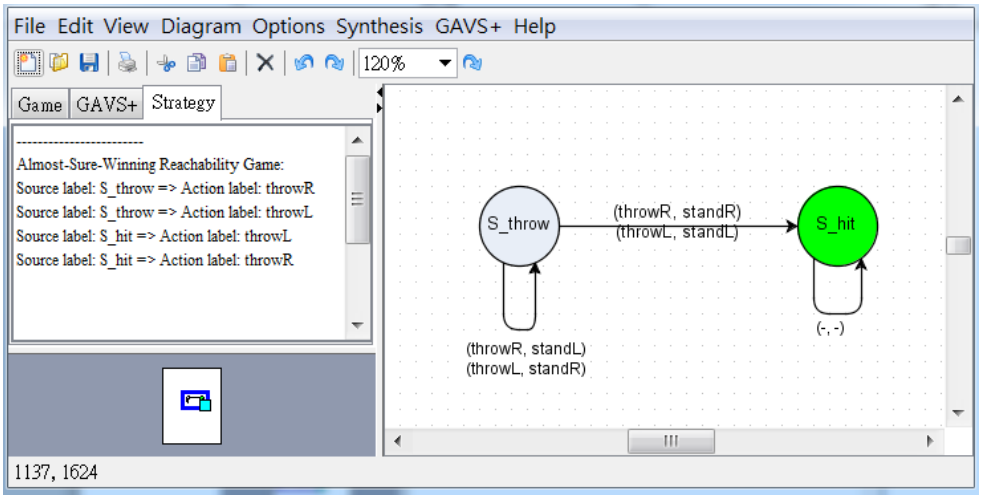

Figure 16. The concurrent reachability game (left-or-right) described in [5] and the generated strategy (almost-sure winning).

- In the edge labeling, `(act1, act2)` means that player-0 uses action act1 and player-1 selects act2. The symbol `(-,-)` is designed for user convenience; the engine will generate all possible combinations of action pairs in its internal representation.
- For concurrent reachability games, use graph labeling (mark the target state in green) to create the specification.

[Limit-sure winning] Here we summarize solving games with limit-sure winning strategies.

- For limit-sure winning strategies, the user should specify the ε value for limit-sure winning, which is offered by GAVS+ using an additional dialog.
- For the generated strategy, given a winning position:
  1) If a strategy is labeled with probability value, then it is executed based on the probability value. In Figure 17, the randomized strategy for state S_{hide} is to perform action **"run"** with probability 0.1 and action **"hide"** with probability $(1 - 0.1)$.
  2) Otherwise, all other strategies should perform uniformly at random with the remaining probability from (1).
- Notice that for limit-sure winning, currently due to our algorithm design, if a state is a goal state, it will **not** be listed/reported on the strategy panel.

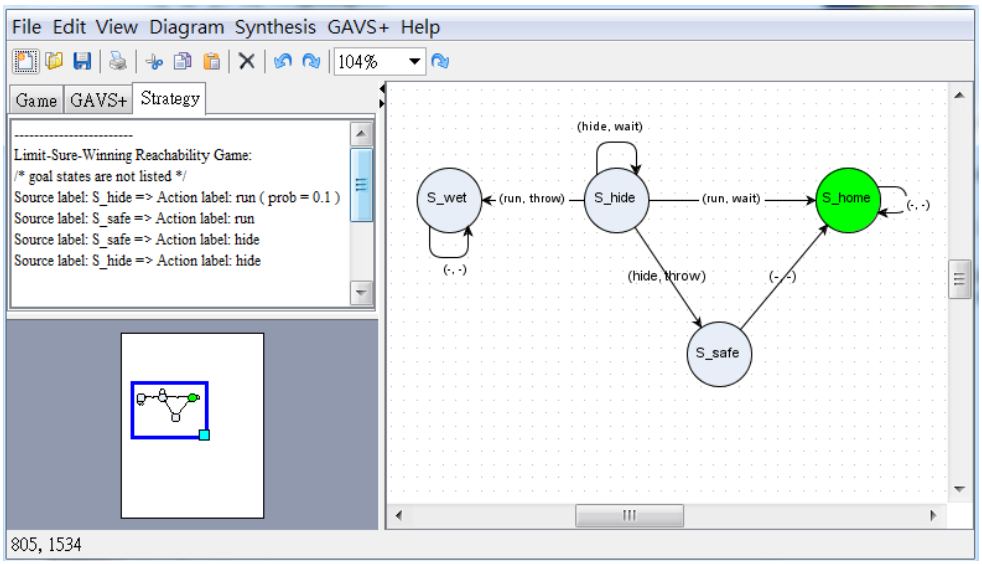

Figure 17. The concurrent reachability game (hide-and-run) described in [5] and the generated strategy (limit-sure winning).

D. Extensions for games of imperfect information

In this section, we discuss an extension for two-player, turn-based games where *imperfect information* is imposed on player 0. Intuitively, imperfect information refers to the phenomenon that player 0 should make decision with the

constraint that he is unable to precisely know his position: instead of position, an *observation* is assigned to player 0. A motivating example for games of imperfect information can be found when synthesizing a controller connected to a digital sensor which reads an analog signal. As no concrete analog value is offered to the controller, the value obtained from the digital sensor (with finite precision) can be viewed as an observation.

1) Definition: A *game of imperfect information* extends from a two-player, turn-based game graph $A = (V_0 \uplus V_1, E)$, where player 0 is unaware of his position with absolute precision. The imprecision is defined by an *observation set* (Obs, γ) where $\gamma : \mathsf{Obs} \to 2^{V_0}$ such that $\forall v \in V_0 \cdot \exists \mathsf{obs} \in \mathsf{Obs} : v \in \gamma(\mathsf{obs})$ (Obs is a finite set of *identifiers*). During a play, when reaching a vertex $v \in V_0$, an arbitrary observation accompanied with v will be assigned to player 0; he is only aware of the observation but not the location. As the location is not known with precision, the successors are imprecise as well. Thus edges for player 0 are labeled with elements in a set Σ of *actions*. A strategy for player 0 should be *observation-based*: it means that the strategy is a function $f : \mathsf{Obs}^+ \to \Sigma$ from the history of observations to actions. For all player 1's edges, a unique label u is used.

2) Example: We give a simple example to explain the concept. For Figure 18, consider the corresponding arena of imperfect information $A = ((V_0 \uplus V_1, E), (Obs, \gamma), \Sigma)$.

- $V_0 = \{v_0, v_2, v_4, v_{risk}\}$, $V_1 = \{v_1, v_3, v_{freeze}, v_{burn}\}$.
- $\Sigma = \{heat, cool\}$.
- $E = \{(v_0, heat, v_1), (v_0, cool, v_{freeze}), (v_1, u, v_0), (v_1, u, v_2), (v_2, cool, v_1), (v_2, heat, v_3), (v_3, u, v_2), (v_3, u, v_4), (v_4, cool, v_3), (v_4, heat, v_{burn}), (v_{burn}, u, v_{risk}), (v_{freeze}, u, v_{risk})\}$.
- $\mathsf{Obs} = \{cold, hot, danger\}$.
- $\gamma(cold) = \{v_0, v_2\}, \gamma(hot) = \{v_2, v_4\}, \gamma(danger) = \{v_{risk}\}$.

Assume that a game starts with the initial location v_2. Based on the received observation (in Figure 18 a value within the set $\{\mathbf{cold}, \mathbf{hot}\}$; remember the application concerning the reading of a digital sensor), the controller should decide either to heat up ($heat$) or to cool down ($cool$) the system. The goal of the game is to create an observation-based strategy such that the system never reaches the risk state v_{risk}.

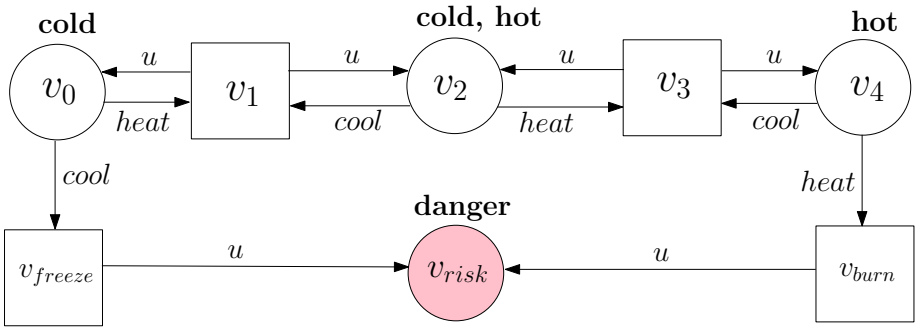

Figure 18. A game of imperfect information for temperature control.

3) Construction and synthesis in GAVS+: In the implementation of GAVS+, the game graph looks slightly different:

- Each node is labeled with a set of observation identifiers. For example, the vertex `V_burn:2` has its name as `V_burn` and observation identifier `2`.
  - Following Figure 18, we interpret observation `2` as `hot`, and observation `1` as `cold`.
  - Location `V2` is labeled with "`:INI`", indicating that it is the initial location.
- Actions are split into *controllable and uncontrollable actions*.
  - In Figure 19, the self-loop of `V_burn` is labeled with "`u:1`", where `u` is the action symbol, and `1` indicates that it is an uncontrollable action. Similarly, for the edge labeled "`heat:0`", it is a controllable action with action symbol `heat`.

The execution proceeds by first receiving an observation. Then player-0 chooses an controllable action, followed by the environment who arbitrarily executes an uncontrollable action. It is not difficult to observe that starting with vertex labeled `V2` with two possible observations `1` and `2` (i.e., `cold` and `hot`), Figure 19 models the same temperature control system as Figure 18. At locations `V_burn` and `V_freeze`, player-0 is unable to perform a move, and thus are considered as risk states.

On the strategy panel in Figure 19, we can examine the observation-based strategy (a finite state machine) generated by GAVS+.

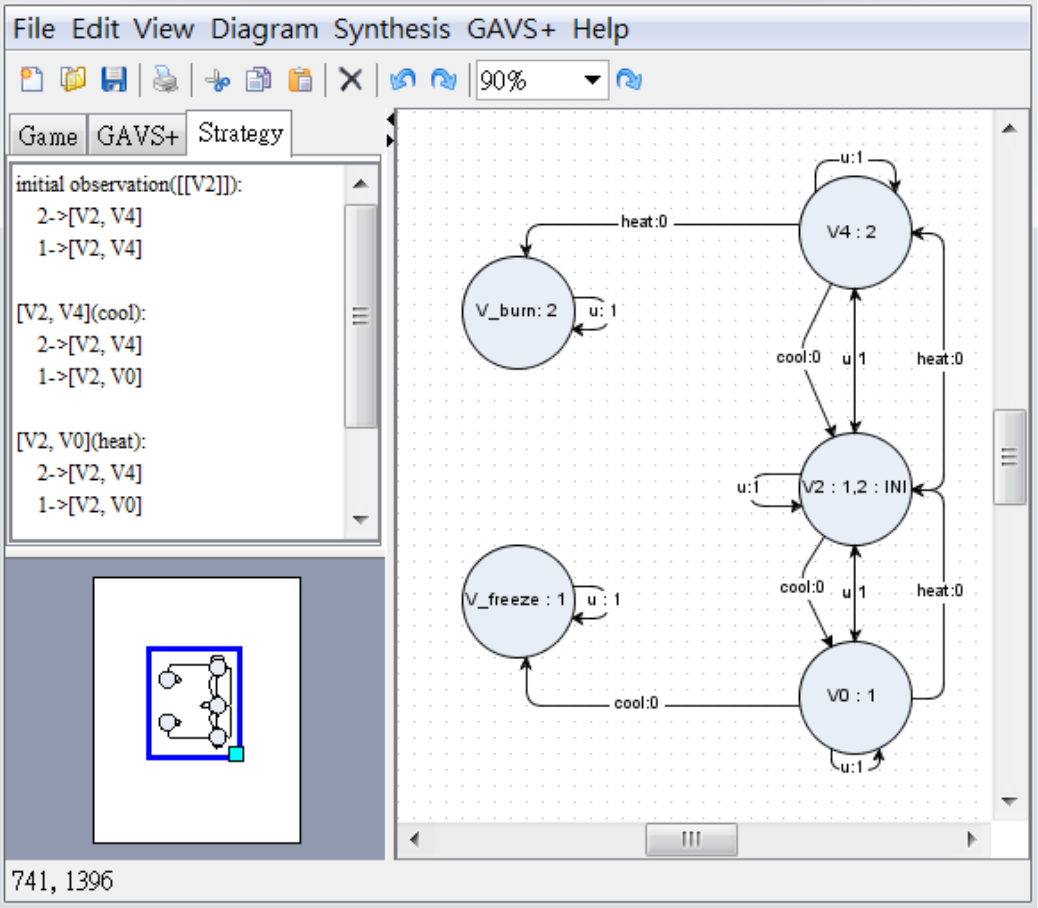

Figure 19. Simple temperature control in Figure 18 modeled using GAVS+.

- Initially, starting with vertex `V2`, when the observation returns `1` or `2`, the text "`1->[V2,V4]`" indicates that player-0 is certain to be in the vertex set {`V2`, `V4`}.
- Then "`[V2, V4](cool)`" indicates that as player-0 is certain to be in the vertex set {`V2`, `V4`}, he should play with the action symbol `cool`. If player-0 continues the play by following the strategy, locations `V_burn` and `V_freeze` are never reached.

E. Extensions for distributed systems: reachability game (experimental)

Distributed games [9] are games formulating multiple processes with no interactions among themselves but only with the environment. Although the problem is undecidable in general [9], [8], finding a distributed positional strategy for player-0 (for reachability) is still practical. As this problem is NP-complete, we translate the problem into SAT and use SAT solvers to generate the strategy.

1) Construction of distributed games: As a distributed game is derived from several local games, GAVS+ supports the drawing of local games using boxes. The box can be selected from the "**GAVS+**" panel. Notice that currently, we **EXPLICITLY** define in our engine the set of environment moves as the **FREE PRODUCT** of all environment moves in the created game. This is an additional constraint to avoid constructing the product game graph. We will release this constraint soon.

Example Open **`/GAVS_Plus_Testcase/DG/DG2.mxe`**, the corresponding distributed game is indicated in Figure 20. To label a vertex in the game as initial state, attach "**`:INI`**" on the vertex, similar to the example.

- Execute the engine by selecting **`GAVS+ -> Distributed Game -> Bounded-SAT Positional P0 Strategy (P0 count on individual process move)`**[1] (Figure 21). A pop up window requires you to give the specification file.
  - Select **`/GAVS_Plus_Testcase/DG/DG2.spe`**. The specification contains a single line "**`Distributed1.r2, Distributed2.q3`**", meaning that the goal is to reach the combined state `(r2, q3)` from the initial state

[1]For this option, we do not restrict that in a control location, each vertex which belongs to P0 in the local game should make a step. We only ensure that one of them should proceed, and when the game graph reaches the state where all local games are in the environment (P1) position, the global move of environment executes.

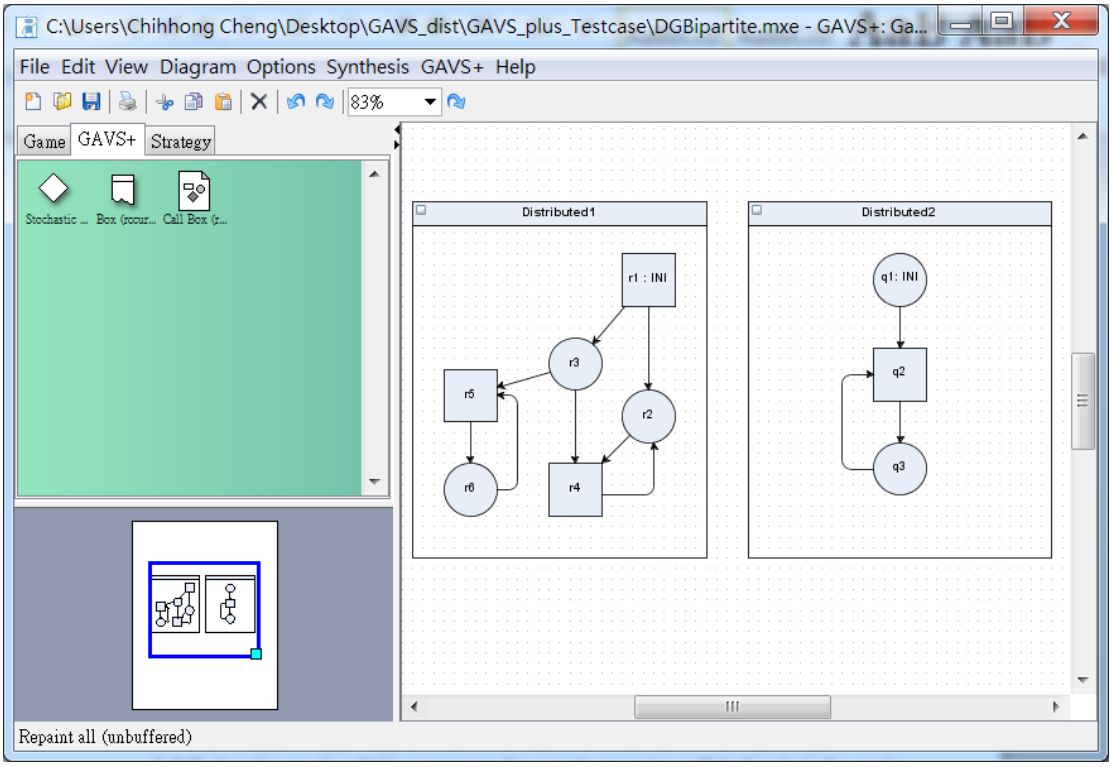

Figure 20. Constructing distributed games.

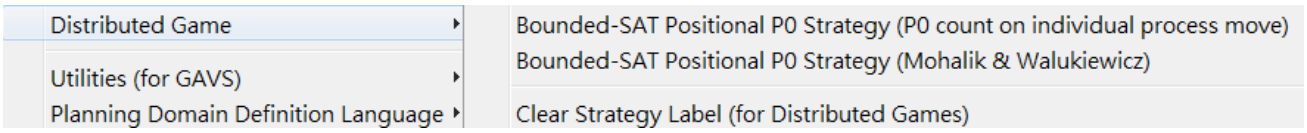

Figure 21. The menu bar for executing distributed games.

`(r1, q1)`.

- Input the number of unrolling images. In this example, select values greater or equal to 6.

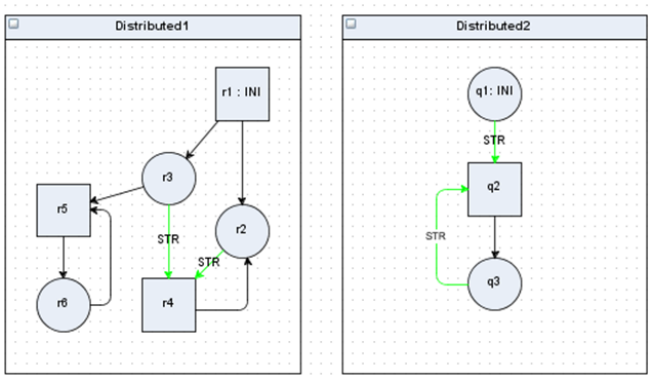

Figure 22. The generated distributed strategy for Figure 20.

- The result of the positional strategy (if there exists one) is indicated on the arena.
  - **[Analysis]** Starting from the initial state, q1 first moves to q2. Then (r1, q2) performs a move to either (r3, q3) or (r2, q3). From (r3, q3), after two steps the system reaches (r4, q2). The environment is unavoidable to move to (r2, q3). In this way, both possible plays based on the positional strategy end up reaching the goal. The maximum steps required to reach (r2, q3) is 5 steps.
- To clear the strategy label, please select **`GAVS+ -> Distributed Game -> Clear strategy labels (for DGs)`**. This is used to ensure that the strategy label in the box is cleared.

V. Behavioral-level synthesis using the Planning Domain Description Language (PDDL)

A. A brief introduction to PDDL

It is important to connect the use of games with concrete application domains. In GAVS+, we establish this connection by offering the translation scheme from PDDL [6] to symbolic games. *Planning Domain Description Language (PDDL)* is the standard language used for behavioral-level planning in artificial intelligence and automation. A planning algorithm generates a sequence of actions from the initial configuration to the goal configuration (or other criteria, e.g., repeated behavior). It can naturally be viewed as trying to find the path for reachability, which can be done using symbolic techniques.

An instance under planning consists of two parts, namely the domain and the problem.

- The ***domain*** contains parameterized system descriptions, including predicates and actions.
- The ***problem*** contains objects, the initial configuration and the goal.

GAVS+ uses the library PDDL4J (license compatible with GPL v3) to perform parsing of the domain and the problem in PDDL format. The object is then translated to symbolic representations of transition systems or games, depending on the description of the domain.

- **[Notice]** To treat the PDDL domain as games, a binary predicate **`POTRAN`** should be used on all control actions, and (**`not POTRAN`**) should be applied on all environment actions.
- **[Notice]** In GAVS+, the supported model for game and planning is the restricted version of Action Description Language (ADL), which is based on the extension of the STRIPS format with one difference:
  1) It follows the open-environment assumption, meaning that in the precondition of an action, if a propositional variable is not specified in the precondition, it can be executed with value either true or false; the STRIPS applies the closed-environment assumption: if a propositional variable is not specified in the precondition, it can only be executed when the precondition is false.
  2) Also, in the initial condition, if a propositional variable is not specified, it is considered to be false.
  3) In the PDDL domain file, the requirement is specified with the following form: (`:requirements :strips :negative-preconditions`)
  4) For details, we refer readers to the book "Artificial Intelligence: the Modern Approach".

[Notice] To execute this functionality, huge memory may be required due to the construction of the BDD. Please set up the parameter option **`-Xmx512m`** (using memory 512Mb) to guarantee that it is possible to use more memory than default. For complex examples (e.g., production line examples), we suggest to use the memory option **`-Xmx2000m`** or higher.

B. Monkey and the (swinging) banana in AI experiments

In this example, we first consider a classical planning problem of a monkey retrieving the banana. There are positions (e.g., p1, p2, p3 and p4) where a box, a knife, as well as the monkey can be placed arbitrarily based on the initial configuration. The banana is hanging on the top of one location, and in order to get the banana, the monkey shall grasp the knife, push the box to dedicated positions, climb up the box, and finally, cut the banana. Figure 23 illustrates the experiment setup, and Table II describes all possible actions for the monkey.

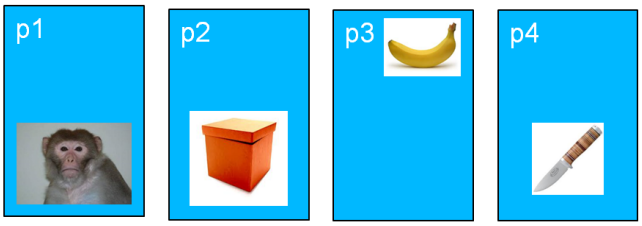

Figure 23. An illustration for the monkey experiment in AI.

1) Solving planning problem using GAVS+: To solve the "planning" problem, execute **`GAVS+ -> PDDL -> Solve PDDL using Symbolic Forward Reachability`**. Two pop-out windows will show up for users to select the domain and the problem.

- For the domain, select **`/pddl/planning/monkey/monkey.pddl`**
- For the problem, select **`/pddl/planning/monkey/pb1.pddl`**

Under this problem setting, we have monkey@p1, box@p2, banana@p3, knife@p4. The result of the planning will be shown on the **`Strategy`** panel (see Figure 25 for reference), which is identical to our expectations[2].

[2]In fact, the symbolic exploration algorithm returns the shortest action sequence.

Table II
ACTIONS DEFINED IN THE PDDL DOMAIN (MONKEY EXPERIMENT)

Action	Parameter	Intuitive meaning
go-to	?pos1 ?pos2	monkey moves from ?pos1 to ?pos2
climb	?pos1	monkey climbs up to the box at position ?pos1
push-box	?pos1 ?pos2	monkey pushes the box from ?pos1 to ?pos2
get-knife	?pos1	monkey picks the knife located at position ?pos1
pick-glass	?pos1	monkey picks the glass located at position ?pos1
cut-banana	?pos1	monkey gets the banana at ?pos1 by using the knife and standing on the box
drink-water	?pos1	monkey drinks water at ?pos1 by using the glass and standing on the box

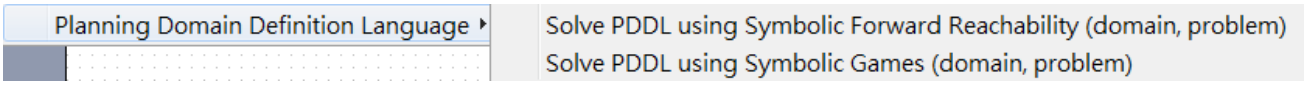

Figure 24. The menu bar to perform analysis over PDDL models.

2) Solving games using GAVS+: Second, we consider a simple extension where the banana can be brought to other places by the experimenter. However, the maximum number of movements is limited to one. As now it is a game setting, planning does not generate the result. We must use the synthesis engine to analyze the problem.

1) **[Solving synthesis problem using GAVS+: First example]** To solve the "planning" problem, execute **`GAVS+ -> PDDL -> Solve PDDL using Symbolic Games -> Reachability`**. Two pop-out windows will show up for users to select the domain and the problem.
   - For the domain, select **`/pddl/synthesis/monkey/monkey.pddl`**
     - Observe the content in monkey.pddl. It is different from the previous one: Existing actions are added with the precondition **`P0TRAN`**" and the postcondition **`(not (P0TRAN))`**.
     - Two additional actions **`BLOW`** and **`STAY`** are listed for the movement of banana. These actions are actions are with the precondition **`(not (P0TRAN))`** and the postcondition **`P0TRAN`**.
   - For the problem, select **`/pddl/synthesis/monkey/pb1.pddl`**
   - Then select "Reactive" for output strategy.
   - *Result* GAVS+ reports a negative result: it is impossible to have a strategy which guarantees that the monkey can get the banana from the initial state!
2) **[Solving synthesis problem using GAVS+: Second example]** In the second example, we increase the ability of the monkey by specifying more actions (similar to fault-tolerant behavior patterns).
   - For the domain, select **`/pddl/synthesis/monkeyUpAndDown/monkeyUpDown.pddl`**.
     - Observe the content in `monkeyUpDown.pddl`. It is different from the previous one: One additional action **`CLIMB-DOWN`** is added; the monkey now can climb down from the box.
   - For the problem, select **`/pddl/synthesis/monkeyUpAndDown/pb1.pddl`**.
   - Then select "Reactive" for output strategy.
   - *Result* GAVS+ reports a positive result with the choice of outputting the strategy.
     - Intuitively, the strategy ensures that the monkey climbs down the box when (a) it does not have a knife or (b) the banana is not in that position.
     - For details concerning the interpretation of result, we refer readers to the GAVS+ website.

C. Fault-tolerant task planning for humanoids

We consider the scenario of a working robot having two humanoid arms (for concepts and behavior of the arm, see http://www6.in.tum.de/Main/ResearchEccerobot for details). When an arm is out-of-service, the tension over the artificial muscle is freed, and the object under grasp falls down to the ground. By modeling the domain and problem using PDDL, we synthesize strategies to perform tasks and *resist the potential loss of one arm.*

1) Solving planning problem using GAVS+ with the gripper example: To solve the "planning" problem, execute **`GAVS+ -> PDDL -> Solve PDDL using Symbolic Forward Reachability`**. Two pop-out windows will show up for users to select the domain and the problem.

- For the domain, select **`/pddl/planning/gripper/gripper.pddl`**.
- For the problem, select **`/pddl/planning/gripper/pb1.pddl`**.

2) Solving synthesis problem using GAVS+ with the gripper example: To solve the "planning" problem, execute **`GAVS+ -> PDDL -> Solve PDDL using Symbolic Games -> Reachability`**. Two pop-out windows will show up for users to select the domain and the problem.

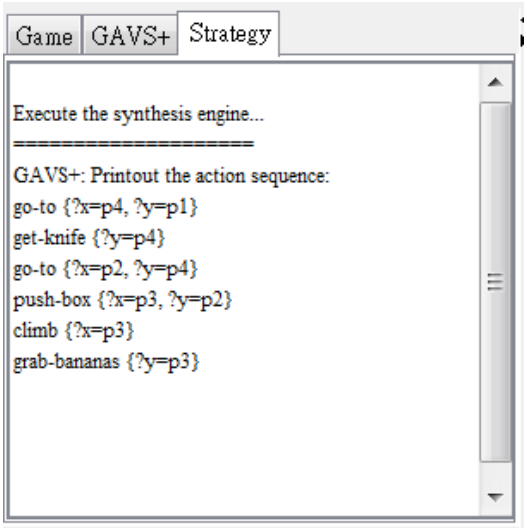

Figure 25. The generated plan for monkey by GAVS+.

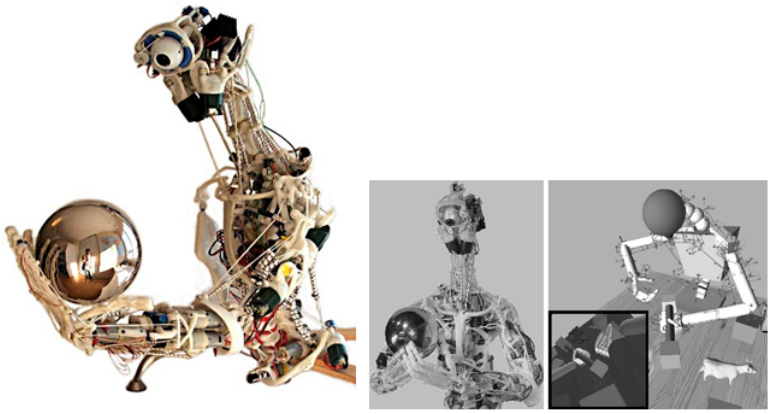

Figure 26. The ECCERobot with two robot arms.

- For the domain, select **/pddl/synthesis/gripper/gripper.pddl**.
- For the problem, select **/pddl/synthesis/gripper/pb1.pddl**.

D. Supported winning conditions in synthesis: using robot navigation as examples

GAVS+ supports synthesizing controllers with many common winning conditions. We explain these winning condition using the following example in robot navigation[3]. Figure 27 illustrates the setup, while the domain and the problem of this example can be found in the package folder **/pddl/synthesis/modified_from_MBP/**. The robot in (a) has four moving options, namely moving `up`, `down`, `left`, and `right`, and the ability to perform such movement depends on the current position (e.g., when a robot is in Storage-room, it is impossible to move up and left). Nevertheless, when moving from the storage room to the right, there exists uncertainty (which is modeled as environment moves) such that the robot may appear in the room `Lab` or `NE_room`.

1) Reachability: For reachability condition, the goal is to reach a set of desired states. E.g., starting with position `Storage-room`, reach `Processing-room`.

- On the menu bar, select `GAVS+ -> Planning Domain Definition Language (PDDL) -> Solve PDDL using Symbolic Games -> Reachability`
  - For the domain, select **robot_navigation_1_GAVS.pddl**
  - For the problem, select **robot_pb1_GAVS.pddl**
- Select "Sequentialized Reactive" for output (you may also select Reactive).

[3]This example is taken and extended from the example in the tool MBP [1], a model-based planner.

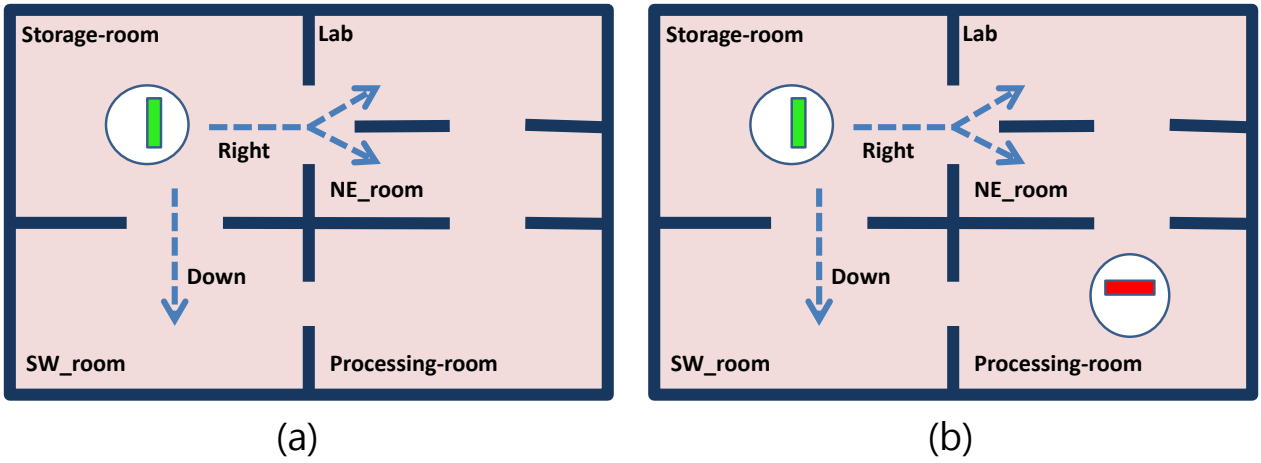

Figure 27. Scenario for single robot navigation (a) and cooperative service between two robots (b).

- When the synthesis is done, the result can be saved as a separate file (view result for sequentialized reactive and reactive).

2) Büchi: For Büchi condition, the goal is to repeatedly reach a set of desired states. E.g., starting with position `Storage-room`, reach `Processing-room` repeatedly.

- On the menu bar, select **`GAVS+ -> Planning Domain Definition Language (PDDL) -> Solve PDDL using Symbolic Games -> Buechi`**
  - For the domain, select **`robot_navigation_1_GAVS.pddl`**
  - For the problem, select **`robot_pb1_GAVS.pddl`**
- When the synthesis is done, the result can be saved as a separate file (view result).

3) Generalized Reactivity: For generalized reactivity, it is the specification of the following form

$$(\square\Diamond p_1 \wedge \ldots \wedge \square\Diamond p_m) \rightarrow (\square\Diamond q_1 \wedge \ldots \wedge \square\Diamond q_n)$$

where "$\square$" and "$\Diamond$" are temporal operators with intuitive meaning of "*always*" and "*in the future*"[4], p_i, q_j are boolean combinations over atomic propositions. Such specification, called Generalized Reactivity[1] conditions, can be efficiently solved in time cubic to the size of specification. E.g., starting with position `Storage-room`, to reach `Storage-room` and `Processing-room` repeatedly can be described in GAVS+ as the following formula **`[]<> true -> []<> (robotposition Processingroom) && []<> (robotposition Storageroom)`**

- On the menu bar, select `GAVS+ -> Planning Domain Definition Language (PDDL) -> Solve PDDL using Symbolic Games -> Generalized Reactivity`
  - For the domain, select **`robot_navigation_1_GAVS.pddl`**
  - For the problem, select **`robot_pb1_GAVS.pddl`**
  - For the GR(1) condition, copy the following (the specification is also available in the problem file) **`[]<> true -> []<> (robotposition Processingroom) && []<> (robotposition Storageroom)`**
- When the synthesis is done, the result can be saved as a separate file (view result).

4) Safety: For safety condition, the goal is to never touch certain states. E.g., consider the scenario in Figure 27(b), where our goal is to create a coordination of service between two robots. To achieve this goal, we can freely design the red robot, and impose the safety constraint on the green robot: the goal is to avoid two robots to be simultaneously in the same room.

E. Quantitative synthesis for performance guarantees

For the behavioral synthesis framework, users may specify appropriate cost bound on each *control action* (i.e., environment actions will always have cost 0), so that provided with a user-specified upper bound, the synthesized controller will guarantee, for all possible runs, to have accumulated cost less than the bound.

[4]Therefore, $\square\Diamond$ represents *infinitely often*.

1) Cost-bound for reachability games: The first example demonstrating the usage of performance guarantees is an example for FESTO Modular Production Systems. The winning condition of this game is to create a controller that successfully process the workpiece, store them appropriately, while the overall cost is within certain bound.

- On the menu bar, select **GAVS+ -> Planning Domain Definition Language (PDDL) -> Solve PDDL using Symbolic Games -> Reachability + Cost bound**
  - For the domain, select **/PDDL/synthesis/FESTO/QuantitativeSynthesis/Quantitative.pddl**. Open the domain with a separate text editor. `(increase (total-cost) 3)` is listed in the effect of the first control action. This means that the cost of using that action will be 3.
  - For the problem, select **/PDDL/synthesis/FESTO/QuantitativeSynthesis/pb.pddl**.
  - Specify the cost bound to be 50. If the bound is set to be 20, there exists no feasible solution. The user is encouraged to manually tune the bound.
- When the synthesis is done, the result can be saved as a separate file.

2) Cost-bound for Goal-or-loop games: Goal-or-loop games define winning conditions such that (1) either the controller needs to guide the play to reach a goal state, or (2) it is allowed to loop forever if environment does not provide required support. When specifying the condition formally using LTL, the specification is like the following:

$$\Box\ (LoopCondition_1 \vee \ldots \vee LoopCondition_k) \vee \Diamond\ GoalCondition$$

We examine an example from SMErobotics, where the specification is to assemble the workpieces when both two pieces are present. Therefore, it is allowed to loop forever if any workpiece is not present. However, we also do not want the robot to do meaningless tasks (such as moving) during its waiting process. Therefore, by setting the cost of `idle-action` to be 0 and the cost of other actions to be 1 (as energy consumption), the generated controller will not consume excessive energy during its waiting process.

- On the menu bar, select `GAVS+ -> Planning Domain Definition Language (PDDL) -> Solve PDDL using Symbolic Games -> Goal-or-loop + Cost bound`
  - For the domain, select **/PDDL/synthesis/SMERobotics/QuantitativeSynthesis/robotics.pddl**.
  - For the problem, select **/PDDL/synthesis/SMERobotics/QuantitativeSynthesis/pb.pddl**.
  - Specify the cost bound to be 10.
  - Enter two looping conditions `(not (present bigobj))` and `(not (present smallobj))`. These two conditions specify the condition that it is allowed to loop when workpieces are not present. Press `cancel` to stop providing looping conditions and start synthesis.
- When the synthesis is done, the result can be saved as a separate file.

F. Guidelines for system modeling

In this section, we list important guidelines concerning system modeling. Modeling deviating from these guidelines may induce errors in symbolic encoding and generate incorrect result for synthesis. Also some modeling tips are explained to speed up the synthesis.

- For `:precondition`, it is suggested to be a conjunction of literals where negations are pushed close to the predicate.
- Currently, the construct of conditional effects (i.e., using the `when` operator in the `:effect` field) is still very unstable. Previously in PDDL, the semantics of (`when` Pre_W *Eff*) means that if Pre_W is true before the action, then effect *Eff* occurs after; this action is feasible even if Pre_W is false (provided that the `:precondition` is evaluated to true) [7]. Pre_W is called a *secondary condition*. In our construct, currently if a secondary condition appears in an action, then the effect will be triggered when both primary and secondary conditions are true. However, the action can not be triggered when only primary condition is true.
- Currently the engine is only applicable with predicates using two parameters. When more than three parameters are used, please modify the domain.
- To increase the speed of the synthesis, several automatic optimization schemes are prebuilt in the engine. One method is to use binary encoding to compact the representation. For instance, for a predicate P with one parameter x of domain C, our engine is able to detect whether at most one of them is evaluated to true. If so, then within the system encoding, binary compaction will be triggered. Notice that this optimization is only applicable with the last parameter. Therefore, please consider a better parameter ordering such that the optimization can be used.

G. Other Examples

For more examples, we refer users to the folder `/pddl/synthesis/`, where the following scenarios are included.

- **(FESTO MPS domain)** This domain describes a production line system under FESTO MPS, where the goal is to store the processed object to different racks based on its color.
- **(Elevator)** This domain describes the request of users for an elevator system.
- **(Model train)** This domain describes the behavior of train and the topology of the track, where the goal is to ensure that a train never derails, and two trains never crash.

VI. Generating code fragments for synthesized strategies (finite-games, push down games)

A. Finite game graph with positional strategy

As a supporting feature, GAVS+ enables to output the generated strategy to executable code fragments, such that it is possible (and easier) to generate a reactive controller based on refining the code. An intuitive way to view the game is to treat it as a prototyping (sketching) of functionalities or behaviors of the system, where in each Player-1 location, the system reads the input from the environment and performs the update.

To invoke the code generation functionality, execute **`GAVS+ -> Utilities (for GAVS) -> Code generation (Java Class)`**. GAVS+ then asks the user to provide the name of the Java class.

Lastly, this functionality is restricted to games having positional strategies.

B. Pushdown games with min-rank strategy (reachability games, Büchi games)

The algorithm for solving pushdown (reachability or Buechi) games generates a data structure called P-automata. P-automata can be used to detect whether starting from the initial state, it is possible to win the game. In the implemented algorithm, during the construction of the P-automata, two additional data structures are created: one is used to record the cost (intuitively, this means how close it is to the target), and another is to record for each edge in the P-automata, the corresponding rewriting rule for player-0.

When a user selects to solve the pushdown game using GAVS+, once when the P-automaton is constructed, an option to serialize the P-automaton and the relevant data structures is provided. By using existing code in the source directory, the user can re-execute the strategy on any Java programs based on deserialization of P-automata (for details concerning deserialization of Java objects, we kindly ask the reader to search in any Java tutorial available in the network). The following code is required for reuse.

- All files in the **`gavs/engine/apds/`** folder: These are used as the basic component of the P-automaton
- For reachability games, the following functions are used in the class **`gavs/engine/APDSEngine.java`**:
  1) `isInitialConfigurationContained()`: Check whether an initial configuration is within
  2) `generateNextMovePositional()`: Generate the next move for player 0
- For Büchi games, the following functions are used in the class **`gavs/engine/APDSEngine.java`**:
  1) `isInitialConfigurationContainedBuechi()`: Check whether an initial configuration is within
  2) `generateNextMoveBuechi()`: Generate the next move for player 0

VII. Invoking GAVS+ on the console using standardized input format

A. Command line options

GAVS+ offers interfaces to invoke the engine from the console without using the GUI. To observe engines supporting the console mode, on the console, execute "**`java -jar GAVS+ -help`**", then the set of available parameters for console execution will be listed, similar to Figure 28.

B. Input file format (two-player turn-based games, MDPs and SSGs)

An example concerning the textural format for games (two-player turned-based, MDP, SSG) in GAVS+ can be found in Figure 29:

We explain the meaning of each field:

- For each line starting with symbols "##", it is treated as comments
- For each vertex in the game, it is equipped with a unique VertexID, which is a positive integer.
- For parity games, VertexID is attached with ":VertexColor". For example, "0:2" is a vertex with ID equals 0 and color equals to 2.
- Each vertex is also assigned with a **`Type`** to indicate its properties. It is also a positive integer, and is predefined in GAVS+. Table 2 summarizes the predefined type.
- For all edges starting from VertexID, we record its destination ID (DestID) and its label (EdgeLabel).
  - For control vertices, each edge label should be "C" (as an indication of controllable)
  - For MDP, edge labels starting from a stochastic vertex should be of the format "`Probability:Reward`", where `Probability` is a double in interval **`[0,1]`**, and `Reward` is a double.
  - For stochastic games, edge labels starting from a stochastic vertex should be of the format "**`Probability`**", which ranges between 0 and 1.

C. Input file format (APDS)

For the file for APDS, as the meaning of each field in Figure 12 is intuitive, we omit the description.

```
************************************************************
GAVS+: Game Arena Visualization and Synthesis (Plus!)
Contact: Chihhong Cheng, TU Munich [chengch@in.tum.de]
************************************************************
Parameter options:
-----------------
1. Normal mode
no parameter : Invoke the GUI
-----------------
2. Engine mode with normal (two-player, turned-based) game graphs
-reach GAMEGRAPH_FILENAME SPEC_FILENAME : Solve the game with reachability criterion
-safety GAMEGRAPH_FILENAME SPEC_FILENAME : Solve the game with safety (co-reachability) criterion
-buechi GAMEGRAPH_FILENAME SPEC_FILENAME : Solve the game with Buechi criterion
-weakpar GAMEGRAPH_FILENAME SPEC_FILENAME : Solve the game with weak-parity criterion
-parity GAMEGRAPH_FILENAME SPEC_FILENAME : Solve the game with parity criterion
-----------------
3. Engine mode with other game types; for those not listed should be invoked using the GUI
-mdpPI GAMEGRAPH_FILENAME GAMMA : Solve the MDP using policy iteration & discount factor GAMMA
-mdpVI GAMEGRAPH_FILENAME GAMMA : Solve the MDP using value iteration & discount factor GAMMA
-mdpLP GAMEGRAPH_FILENAME GAMMA : Solve the MDP using linear programming & discount factor GAMMA
-ssgVI GAMEGRAPH_FILENAME : Solve the simple stochastic game using Shapley (value iteration)
-ssgPI GAMEGRAPH_FILENAME : Solve the simple stochastic game using Hoffman-Karp (policy iteration)
-ssgRandomPI GAMEGRAPH_FILENAME : Solve the simple stochastic game using randomized Hoffman-Karp
-----------------
4. Engine mode with linking to PDDL files
-PDDLplan DOMAIN_FILENAME PROBLEM_FILENAME : Generate a plan (action sequence) for the PDDL with the input
containing domain and problem (no environment move)
-PDDLsyn DOMAIN_FILENAME PROBLEM_FILENAME : Generate a strategy (state machine) for the PDDL with the input
containing domain and problem (with environment move)
```

Figure 28. The help menu in the console mode of GAVS+.

```
## Comments used in the .game file
## VertexID(:VertexColor, optional) Type DestID1 EdgeLabel1 DestID2 EdgeLabel2
...
0 3 0 b
1 4 1 a 0 c
2 3
3 2 2 b 1 d 0 a
```

Figure 29. The textual file format for games (except APDS) in GAVS+.

VIII. Designing and contributing your algorithms with GAVS+

A. Designing your algorithm with GAVS+

GAVS+ is released under GPLv3 for the purpose of provoking an easy extension of the software and a joint collaboration in the field of algorithmic games. Here we list out features for the ease of extension:

- To avoid the access of the graph explicitly, in our implementation we first translate the graph model to a pure mathematical model **`GameArena.java`**, which is easy to manipulate. For users who are interested in designing algorithms, the mathematical model offers a simple starting point: once when the algorithm is designed based on the model, it is easy to redirect the result to the graph model. The general method is as follows:
  1) Modify **`/swing/editor/EditorMenuBar.java`** to include the choice of executing your algorithm.
  2) Modify **`/swing/resources/editor.properties`** to include appropriate descriptions for your menu item.
  3) If your algorithm uses games predefined in GAVS+, then check the relevant action specified in **`/swing/editor/EditorActions.java`** and insert appropriate mechanisms to redirect the game model to the new algorithm.

Game type (visualization)	Vertex Type	Edge Label
Two-player, turn-based finite games	GAME_INITIAL_CONTROL = 1 GAME_INITIAL_PLANT = 2 GAME_NONINITIAL_CONTROL = 3 GAME_NONINITIAL_PLANT = 4	Control: C Plant: P
Markov Decision Process	MDP_GAME_CONTROL = 1 MDP_GAME_STOCHASTIC = 2	Control: C Stochastic: Probability:Reward
(Simple) Stochastic Games	SSG_INITIAL_CONTROL = 1 SSG_INITIAL_PLANT = 2; SSG_NONINITIAL_CONTROL = 3 SSG_NONINITIAL_PLANT = 4 SSG_INITIAL_STOCHASTIC = 5 SSG_NONINITIAL_STOCHASTIC = 6 For P0Sink, attach the vertex with color 1 For P1Sink, attach the vertex with color 2 For others, attach the vertex with color 0	Control: C Plant: P Stochastic:S

Figure 30. The summary for the vertex type specified in GAVS+ based on specific games.

4) The data structure `HashMap<String, String> vertexID_Name_Map` and `name_vertexID_Map` can be used to connect the internal representation and the graphical representation.

- Also, as we provide the translation scheme from PDDL to games, it is very easy to replicate the process. For example, it is possible to solve planning problems in PDDL with MDP solvers.

B. Contributing your algorithms to GAVS+

For researchers using GAVS+ as their development tool, it is possible (and highly welcome) to contribute their work to the official release of GAVS+. In the GAVS+ website, we will specify contributors with their implemented modules, similar to the Ptolemy II project in UC Berkeley[5]. Once when a new algorithm is received, we will release a beta version containing this improvement. A major revision will be released later. Please contact Chih-Hong Cheng (cheng.chihhong@gmail.com) for further details.

REFERENCES

[1] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Mbp: a model based planner. In *IJCAI-2001 Workshop on Planning under Uncertainty and Incomplete Information (PRO-2)*, 2001.

[2] T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP'02)*, volume 2382 of *LNCS*, pages 704–715. Springer-Verlag, 2002.

[3] C.-H. Cheng, C. Buckl, M. Luttenberger, and A. Knoll. GAVS: Game arena visualization and synthesis. In *Proceedings of the 8th International Symposium on Automated Technology for Verification and Analysis (ATVA'10)*, volume 6252 of *LNCS*, pages 347–352. Springer-Verlag, 2010.

[4] A. Condon. On algorithms for simple stochastic games. *Advances in computational complexity theory*, 13:51–73, 1993.

[5] L. De Alfaro, T. Henzinger, and O. Kupferman. Concurrent reachability games. *Theoretical Computer Science*, 386(3):188–217, 2007.

[6] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(1):61–124, 2003.

[7] M. Ghallab, C. Aeronautiques, C. Isi, S. Penberthy, D. Smith, Y. Sun, and D. Weld. PDDL-the planning domain definition language. Technical Report CVC TR-98003/DCS TR-1165, Yale Center for Computer Vision and Control, Oct 1998.

[8] D. Janin. On the (high) undecidability of distributed synthesis problems. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'07)*, volume 4362 of *LNCS*, pages 320–329. Springer-Verlag, 2007.

[9] S. Mohalik and I. Walukiewicz. Distributed games. In *Proceedings of the 23rd International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, volume 2914 of *LNCS*, pages 338–351. Springer-Verlag, 2003.

[5]http://ptolemy.eecs.berkeley.edu

[10] L. Shapley. Stochastic games. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 39, page 1095. National Academy of Sciences, 1953.

[11] C. White and D. White. Markov decision processes. *European Journal of Operational Research*, 39(1):1–16, 1989.