User manual T-Beg: A Tool for Behavioural Equivalence Games*

Barbara König[1], Christina Mika-Michalski[1], and Lutz Schröder[2]

[1] University of Duisburg-Essen, Germany
[2] Friedrich-Alexander-Universität Erlangen-Nŕnberg
{barbara_koenig,christina.mika-michalski}@uni-due.de,
lutz.schroeder@fau.de

1 Introduction

Behavioural equivalences can be characterized via bisimulation, modal logics, and spoiler-duplicator games. In T-Beg we work in the general setting of coalgebra and focus on generic algorithms for computing the winning strategies of both players in a bisimulation game. The winning strategy of the spoiler (if it exists) is then transformed into a modal formula that distinguishes the given non-bisimilar states. The modalities required for the formula are also synthesized on-the-fly, and we present a recipe for re-coding the formula with different modalities, given a separating set of predicate liftings (for theoretical details we refer to [6, 5]).

In the next section we give an overview of the tool and the main design decisions. After that we explain based on an example how to use the tool including a brief description how T-Beg has been tested. Finally, we give an outlook on the optimization of T-Beg.

2 T-Beg: A Generic Tool for Games and the Construction of Distinguishing Formulas

2.1 Overview

A tool for playing bisimulation games is useful for teaching, for illustrating examples in talks, for case studies and in general for interaction with the user. There are already available tools, providing visual feedback to help the user to understand why two states are (not) bisimilar, such as The Bisimulation Game Game[3] or Bisimulation Games Tools[4] [3]. Both games are designed for labelled transition systems and [3] also covers branching bisimulation.

Our tool T-Beg goes beyond labelled transition system and allows to treat coalgebras in general (under the restrictions that we impose), that is, we exploit the categorical view to create a generic tool.

* Supported by the project BEMEGA funded by the DFG.

[3] http://www.brics.dk/bisim/

[4] https://www.jeroenkeiren.nl/blog/on-games-and-simulations/

A generic algorithm given by the coalgebraic game: We will now present the rules for a *behavioural equivalence game* (if you are not familiar with the underlying concepts, we refer to [6, 5]).

At the beginning of a game, two states x, y are given. The aim of the spoiler (S) is to prove that $x \nsim y$, the duplicator (D) attempts to show the opposite.

- **Initial situation:** A coalgebra $\alpha\colon X \to FX$ and two states $x, y \in X$.
- **Step 1:** S chooses $s \in \{x, y\}$ and a predicate $p_1\colon X \to 2$.
- **Step 2:** D takes $t \in \{x, y\}\backslash\{s\}$ if $x \neq y$ and $t = s$ otherwise and has to answer with a predicate $p_2\colon X \to 2$ satisfying $Fp_1(\alpha(s)) \leq^F Fp_2(\alpha(t))$.
- **Step 3:** S chooses p_i with $i \in \{1, 2\}$ and some state $x' \in X$ with $p_i(x') = 1$.
- **Step 4:** D chooses some state $y' \in X$ with $p_j(y') = 1$ where $i \neq j$.

As shown in [5], the coalgebraic game defined in [6] provides us with a generic algorithm to compute the winning strategies and distinguishing formulas. This algorithm computes the greatest fixpoint of $\mathcal{F}_\alpha\colon Eq(X) \to Eq(X)$ on equivalence relations (where $E(R)$ denotes the set of all equivalence classes over R, F is a functor representing the branching type and χ_P denotes the characteristic function of a subset $P \subseteq X$):

$$\mathcal{F}_\alpha(R) = \{(x, y) \in R \mid \forall P \in E(R)\colon F\chi_P(\alpha(x)) = F\chi_P(\alpha(y))\}$$

In Algorithm 1 $I(x, y)$ denotes the first index where x, y are separated in the fixpoint iteration of $\mathcal{F}_\alpha$. More precisely, i is the least least index i such that $(x, y) \notin \mathcal{F}^i_\alpha(X \times X)$. If two states x, y are never separated, i.e., $(x, y) \in \nu\mathcal{F}_\alpha$ we set $I(x, y) = \infty$.

The second component T tells the spoiler what to play in Step 2. In particular whenever $T(x, y) = (s, P_1)$, S will play s and $p_1 = \chi_{P_1}$.

Such a winning strategy for the spoiler can be computed during the fixpoint iteration, see Algorithm 1. (for details we refer to [5]).

Generic in use: The user can either slip into the role of the spoiler or of the duplicator, playing on some coalgebra against the computer. The tool computes the winning strategy (if any) and follows this winning strategy if possible. We have also implemented the construction of the distinguishing formula for two non-bisimilar states.

The genericity over the functor is in practice achieved as follows: The user either selects an existing functor F, or implements his/her own functor by providing the code of one class with nine methods (explained below). Everything else, such as embedding the functor into the game and the visualization are automatically handled by T-Beg. In the case of weighted systems, T-Beg even handles the creation of the graphical representation.

Then, he/she enters or loads a coalgebra $\alpha : X \to FX$ (with X finite), stored as *csv* (comma separated value) file. Now the user can switch to the game view and start the game by choosing one of the two roles (spoiler or duplicator) and selecting a pair of states (x, y), based on the the visual graph representation.

Algorithm 1 Computation of $\nu\mathcal{F}_\alpha$ and the winning strategy of the spoiler

procedure COMPUTE THE WINNING MOVES FOR S
  for all $(x, y) \in X \times X$ **do**
    $I(x, y) \leftarrow \infty$
  $i \leftarrow 0$, $R_0 \leftarrow X \times X$
  repeat
    $i \leftarrow i + 1$, $R_i \leftarrow R_{i-1}$
    for all $(x, y) \in R_{i-1}$ **do**
      for all $P \in E(R_{i-1})$ **do**
        if $F\chi_P(\alpha(x)) \not\leq^F F\chi_P(\alpha(y))$ **then**
          $T(x, y) \leftarrow (x, P)$, $I(x, y) \leftarrow i$, $R_i \leftarrow R_i \setminus \{(x, y)\}$
        else
          if $F\chi_P(\alpha(y)) \not\leq^F F\chi_P(\alpha(x))$ **then**
            $T(x, y) \leftarrow (y, P)$, $I(x, y) \leftarrow i$, $R_i \leftarrow R_i \setminus \{(x, y)\}$
  until $R_{i-1} = R_i$
return R_i, T, I

Next, the computer takes over the remaining role and the game starts: In the game overview, the user is guided through the steps by using two colors to indicate whether it is spoiler's (violet) or duplicator's (cyan) turn (see Figure 1).

In the case of two non-bisimular states, the tool will display a distinguishing formula at the end of the game.

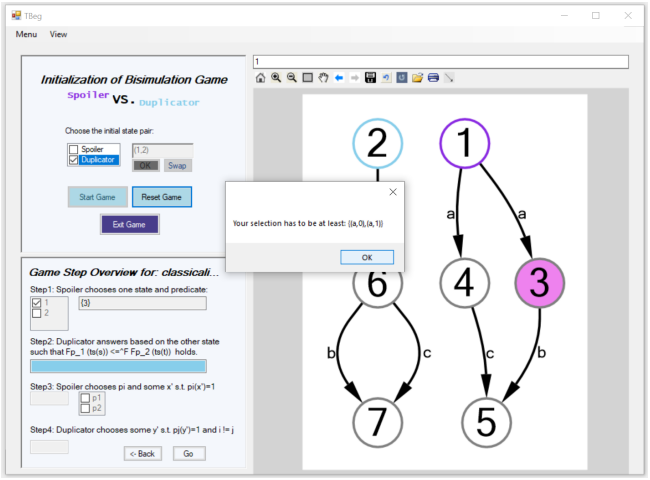

Fig. 1: Screenshot of the graphical user interface with a game being played.

2.2 Design

We now give an overview over the design and the relevant methods within the tool. We will also explain what has to be done in order to integrate a new functor.

T-Beg is a Windows tool offering a complete graphical interface, developed in Microsoft's Visual Studio using *C#*, especially *Generics*. The program is divided into the following five components: Model, View, Controller, Game and Functor. We have chosen *MVC (Model View Controller)* as a modular pattern, so modules can be exchanged. Here we have several $Model\langle T\rangle$ managed by the *Controller*, where the functor in the sense of a *Functor* class, which always implements the *Functor Interface*, is indicated by the parameter $\langle T\rangle$.

While the tool supports more general functors, there is specific support for functors F with $FX = V^{GX}$ where V specifies a semiring and GX is finite (whenever X finite). That is, F describes the branching type of a weighted transition system, where for instance $GX = A \times X + 1$ (introducing labels and termination). Coalgebras are of the form $X \to V^{GX}$ or – via currying – of the form $X \times GX \to V$, which means that they can be represented by $X \times GX$-matrices (matrices with index sets X, GX). In the implementation V is the generic data type of the matrix entries. In the case of the powerset functor we simply have $V = 2$ and $GX = X$.

If the transition system can not simply be modelled as a matrix, there is an optional field that can be used to specify the system, since $Model\langle T\rangle$ calls the user-implemented method to initialize the transition system instance. The implementation of the bisimulation computation described in 1 can be found in $Game\langle T, V\rangle$, representing the core of the tool's architecture.

Functor Interface As mentioned previously, the user has to provide nine methods in order to implement the functor: two are needed for the computation, two for rendering the coalgebra as a graph, one for creating modal logical formulas, another two for loading and saving, and two more for customizing the visual matrix representation.

We would like to emphasize here that the user is not expected to formally implement the functor in the sense of the categorical definition. In particular, we do not need the application of the functor to arrows, but we need methods that evaluate the conditions necessary for the game.

Within *MyFunctor*, which implements the interface $Functor\langle F, V\rangle$, the user defines the data structure F for the branching type of the transition system (e.g., a list or bit vector for powerset functor, or the corresponding function type in the case of the distribution functor). Further, the user specifies the type V that is needed to define the entries of $X \times GX$ (e.g. a double value for a weight or $0, 1$ to indicate the existence of a transition).

Then the following nine methods have to be provided:

$Matrix\langle F, V\rangle InitMatrix(\dots)$**:** This method is necessary to initialize the transition system with the string-based input of the user. The information about the states and the alphabet is provided via an input mask in the form of a

matrix. It is strongly recommended to add a $try\{\}catch\{\}$ block like in the available examples.

bool CheckDuplicatorsConditionStep2(...): given two states x, y and two predicates p_1, p_2, this method checks whether

$$Fp_1(\alpha(x)) \leq^F Fp_2(\alpha(y)).$$

This method is used when playing the game (in Step 2) and in the partition refinement algorithm 1 for the case $p_1 = p_2$.

There are two different scenarios, where this method is used: in the game when the duplicator makes his turn in *Step* 2. The method checks whether the duplicator's move is valid depending on the first move made by the spoiler.

On the other hand, we need this method during the validation of the condition $\leq^F$ described in [5]. There, for a given pair $(x, y) \in X \times X$ and a predicate p over X one has to check whether $Fp(\alpha(x)) \leq^F Fp(\alpha(y))$ holds. If this is not fulfilled, as already described in Algorithm 1, the pair (x, y) is removed from the current relation and $T(x, y), I(x, y)$ are updated accordingly. Note that in this case we have $p_1 = p_2 = p$, i.e., the method is used in the specific case where the two predicates are equal.

TSToGraph(...): This method handles the implementation of the graph-based visualization of the transition system, via an external graph library[5]. In the case of weighted systems the user can trust the default implementation included within the *Model*. In this case, arrows between states and their labels are generated automatically.

GraphToTS(...): This method is used for the other direction, i.e. to derive the transition system from a directed graph given by *Graph*.

(Analogous to the method before, in the case of weighted systems the user can trust the default implementation included within the *Model*.)

string GetModalityToString(...): This method is essential for the automatic generation of the modal logical formulas distinguishing two non-bisimilar states as described in [5]. In each call, the cone modality that results from $F\chi_P(\alpha(s))$ with $T(x, y) = (s, P)$ is converted into a string.

SaveTransitionSystem(...): In order to store a transition system in a *csv* file.

LoadTransitionSystem(...): In order to load a transition system from a *csv* file.

GetRowHeadings(...): T-Beg can visualize a transition system $\alpha : X \to FX$ as a $X \times GX$ matrix within a *DataGrid*. For this purpose, the user needs to specify how the *RowHeaders* are generated automatically.

ReturnRowCount(...): This method returns the number of rows of the matrix representing the coalgebra.

The last four methods are not of theoretical interest, but are needed to increase the usability of the system.

In addition, T-Beg uses a graph library[3], which in turn provides a *GraphEditor* that allows for storing graphs as *MSAGL* files or as *png* and *jpg* files.

[5] `https://www.nuget.org/packages/Microsoft.Msagl.GraphViewerGDI`

3 A small tutorial

We now explain step by step the procedures that are needed to initialize a game instance. Next we present a possible game sequence with the user as the Duplicator. Finally we provide a small overview of how T-Beg has been tested.

After executing T-Beg by double-clicking on *tbeg.exe* a minimally configured Windows Form is presented to the user. A click on *View* at the top left opens a menu containing three items (see Figure 2).

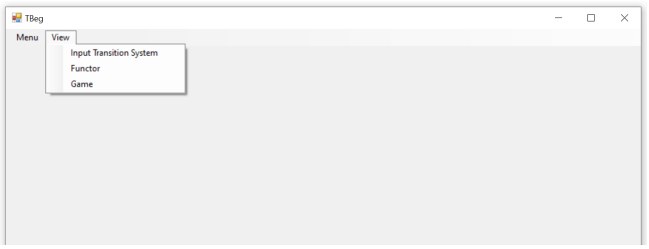

Fig. 2: Getting started with T-Beg.

3.1 How to Use

Initialize a transition system and a game: To proceed please select the view *Input Transition System*. Now, there are two ways to initialize a transition system. First, we consider the more complex way known as "typing manually some input into some mask" illustrated in Figure 3.

Selecting a functor is the first action the user has to perform, this can be done using the *combobox* at the top left of Figure 3. The desired number of states can be set in the edit field to the right of *"Number of states"*. To provide an alphabet the user can utilize the edit field next to the label *"Alphabet"*. Multiple symbols are separated by ",". (Remark: If the user wants to work with a non-labelled transition system then the field has to be blank. The tool will internally treat this with "NL".)

In case of *Powerset* the user has to type "1" at $(x, (a, y))$ to enable an $x \xrightarrow{a} y$ transition. How to use the datagrid in general strongly depends on the implementation of *InitMatrix* described in subsection 2.2.

To continue a name for the system has to be set in the edit field below the *"Enter transition system"* button. After providing a name and pressing the *"Enter transition system"* button, the *"Switch to Graph-View"* button gets enabled. A click on that button opens the game view as seen in Figure 5. (Remark: In case some entries are wrong or the name is already in use, the user will be informed so he or she can check the input.)

Another way of initializing a game is to load some already created and saved transition systems (see Figure 4). For our scenario we selected an existing functor

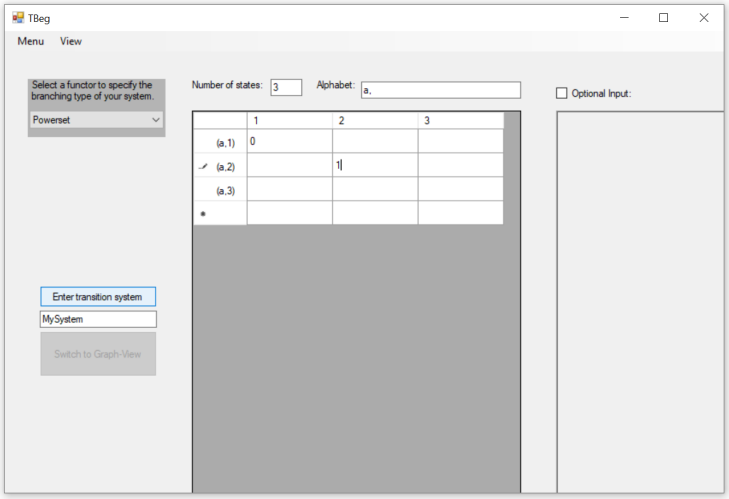

Fig. 3: Manual initialization of a transition system for $F = \mathcal{P}(A \times X)$.

Powerset. Pressing *load* given in the top left corner of the menu T-Beg automatically opens the folder wrt. the selected functor in *bin**Transitionsystems* *Powerset* \. T-Beg also saves the transition system under the provided name into this specific folder pressing *save*.

After a system has been properly initialized in one of the two ways described above and the button *"Switch to Graph-View"* has been enabled and pressed, the user can now start to configure a game over this system.

Play a game as Duplicator: At the beginning two states have to be selected via the mouse. By pressing the *"CTRL"* key more than one state can be selected (*GraphEditor* is provided by [6]). Or you can keep the left mouse button pressed and span an area over the states to be selected. The selection has then to be confirmed using the *"ok"* button illustrated in Figure 6. The *"Swap"* button enables to switch the state pair. (In case $x \nsim y$, T-Beg displays a formula that is satisfied by y and not by x. Otherwise a formula satisfied by x will be provided.)

[6] `https://www.nuget.org/packages/Microsoft.Msagl.GraphViewerGDI`

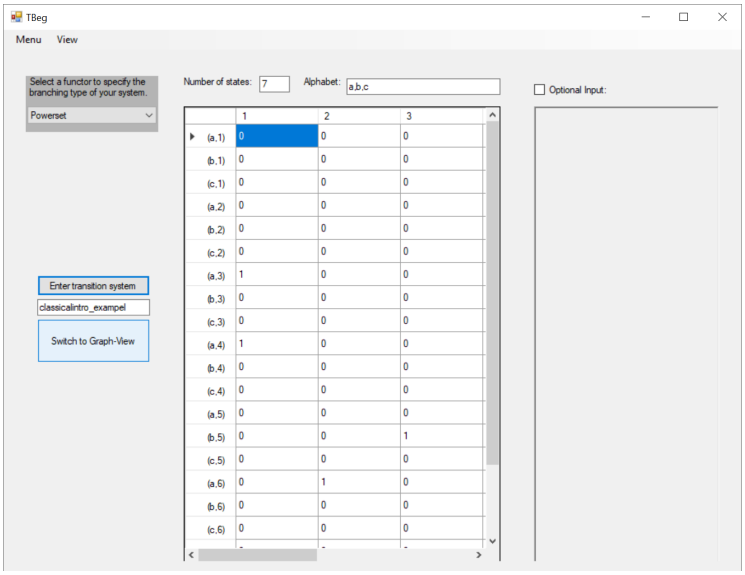

Fig. 4: Loading of *classical_introexample.ts* for $F = \mathcal{P}(A \times X)$.

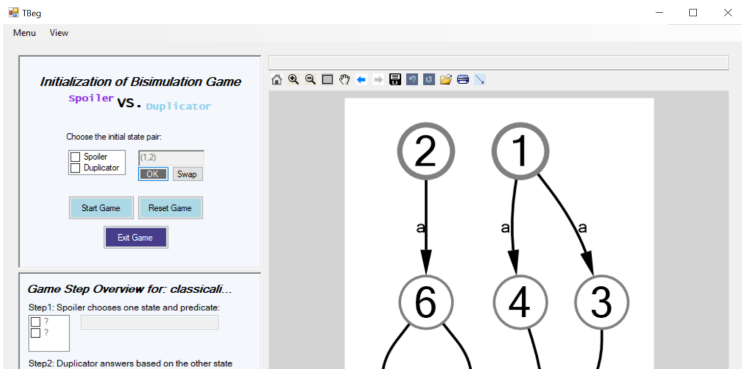

Fig. 6: Selection of two states using the *GraphEditor*

The player mode is set by the two *"checkboxes"* within the initialization panel. To proceed our scenario we selected *"Duplicator"* and started the game by pressing the *"Start game"* button (see Figure 7).

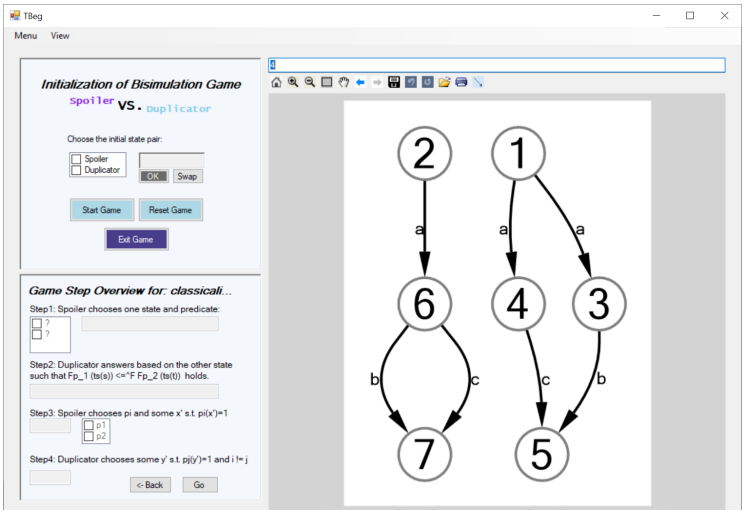

Fig. 5: Just opened game view of *classical_introexample.ts* for $F = \mathcal{P}(A \times X)$.

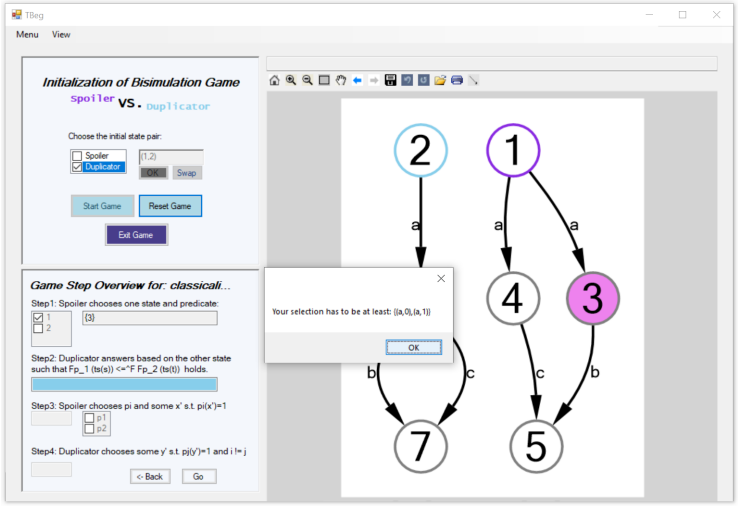

Fig. 7: Move of the *Spoiler* for a currently initialized and started game with user in the role of *Duplicator*.

A note about the button *"Start game"*: If the user has added or removed nodes or transitions using the *GraphEditor* T-Beg will ask the user if she/he wants to play on the previous transition system or the modified. If the user chooses the modified system, T-Beg automatically generates a new name based on the name of the original system and saves it to the corresponding folder (in case all modifications leads to a valid graph wrt. functor). Otherwise the

graph is set back to the previous version. The user has to be aware that graph modifications are disabled during the game.

In Figure 7 the move $T(1,2) = (1,\{3\})$ of the Spoiler is highlighted using violet tones: The selection of state node 1 is visualized by a violet contour and the states captured by the predicate P_1 of the spoiler are filled with a purple color. Furthermore, the duplicator has to continue with state node 2 indicated by a cyan contour.

The information within the *Dialog-Box* in Figure 7 includes the value $Fp_1(\alpha(s))$ in $F\{0,1\}$. This is the minimum value in $F\{0,1\}$ the user has to provide by choosing a predicate P_2 to satisfy the condition $Fp_1(\alpha(s)) \leq^F Fp_2(\alpha(t))$ of *step* 2 in the game (where α is the transition system (ts)). For more details we refer to [5]). The selection of the states takes place here exactly as in the description for the initialization of the game. With the difference that this time the button *"Go"* must be pressed to confirm the choice.

In this specific example given in Figure 7 with $F = \mathcal{P}(A \times X)$ the value $\{(a,0,(a,1))\}$ means that not all a-succesors of state 1 are included in the predicate given by the *Spoiler (*T-Beg*)*.

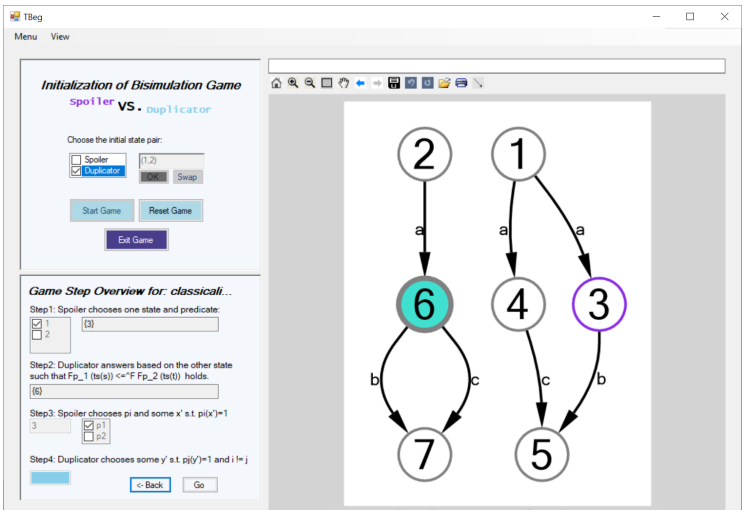

Fig. 8: In *step* 2 of the game the answer of the *Duplicator* (user) is highlighted by the color cyan.

Therefore, the answer $P_2 = \{6\}$ illustrated in Figure 8 evaluating to the value $Fp_2(\alpha(2)) = \{(a,1)\}$ given by the user is sufficient enough. In addition, this answer means that all a-successors of state node 2 are captured. The game is lost if a non-valid answer is provided by the *Duplicator*.

The colouring enables to distinguish the moves of each player. The node that corresponds to the states that are part of the predicate resulted from a Duplicator

move are coloured with some cyan tone (see Figure 8 where the concrete answer is $\{6\}$).

In addition, the next move of the *Spoiler (*T-Beg*)* for *step* 3 is already displayed in the corresponding field and *Checkbox*. Figure 8 shows that the *Spoiler (*T-Beg*)* selected the first predicate and state 3. You can also see that the field of *step* 4 is highlighted in cyan, which signals to the user that it is her/his turn again.

The condition wrt. *step* 4 intends to select a state that is captured in the remaining predicate. Which predicate is available to the user can be seen in the unchecked *checkbox* at *step* 3 (see Figure 8). The user confirms her/his selection with the *"Go"* button. T-Beg initiates the next round if the selected state node is included in the remaining predicate. Otherwise, the user is advised that her/his state node does not meet the condition previously described.

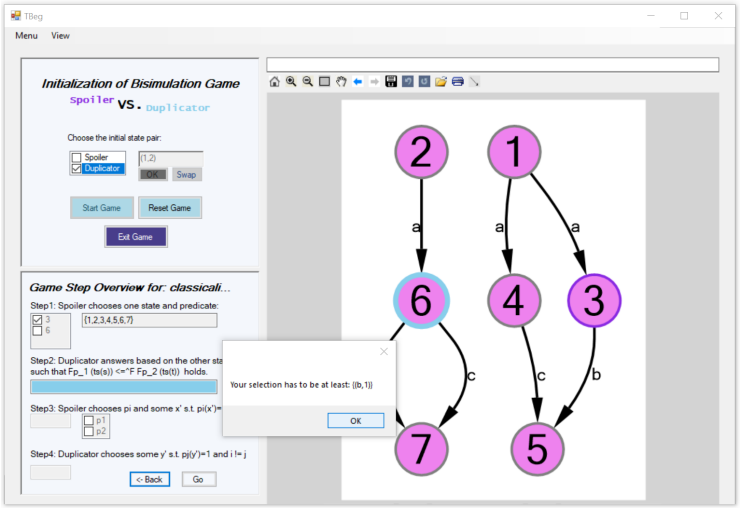

Fig. 9: The next round has been started with the move $(3, \{1, 2, 3, 4, 5, 6, 7\})$ given by the *Spoiler*.

In Figure 9 the *checkboxes* are updated to the selection $(3, 6)$ according to *step* 3 and *step* 4. T-Beg proceeds with the game and therefore the move $(3, \{1, 2, 3, 4, 5, 6, 7\})$ given by the *Spoiler* is already visualized.

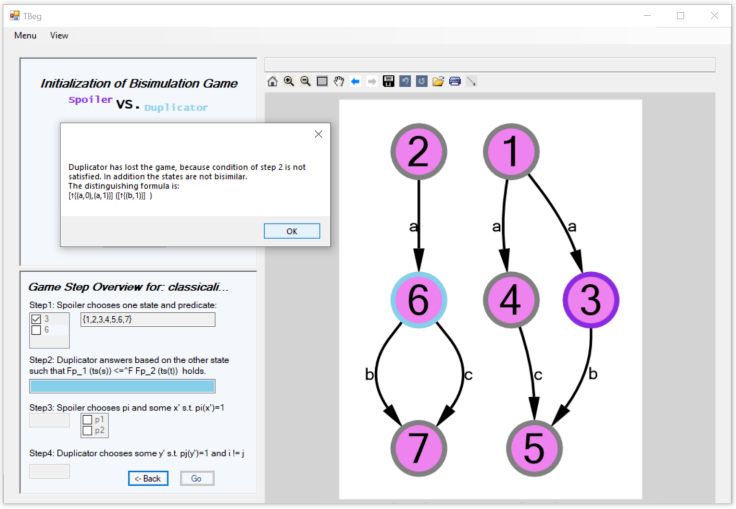

Fig. 10: .

Every time T-Beg in the role of spoiler replies with the entire set of states, as in Figure 9, this means that the duplicator has no way to satisfy the condition of *step* 2. Therefore, in Figure 10 the game leads to an end and the information whether the initial states are bisimilar or not are forwarded to the user. In case $x \nsim y$ a distinguishing formula is also included in this *Dialogbox*.

3.2 Testing Report

So far, T-Beg is an α-prototype tool. The implementation of the bisimulation computation (as the generation of distinguishing formulas) has been tested using hundreds of randomly created transition systems $\alpha : X \to FX$. The focus was to keep the probability of the following errors as low as possible. Mostly, the following errors were identified:

- IndexOutOfRangeException
- ArgumentNullException
- Non terminating procedures

In order to exclude errors that could be caused by the implementation of the functor, the tests were carried out using only one functor (*Powerset*). Therefore, three parameters have been varied: number of states, size of the alphabet and the probability of $x \xrightarrow{a} y$ for each $(x, y) \in X \times X$ and $a \in A$.

The tests for the game-flow were carried out manually and divided into three phases.

- Phase 0: While implementing the methods of each functors a few small transition systems have been used for testing.

- Phase 1: For *Powerset* three examples (classical_introexample, 3levels and conjunction_negation) and two instances (desh_paper and exa_paper) for $(DX + 1)^A$ were intensiveley tested from the spoiler and the duplicator perspective in order to detect unexpected or wrong behaviour within T-Beg. Note that this is very time consuming since the number of possible moves that a user can make depends exponentially on the number of states. However, at the same time the manual tests allows also to verify the feedback to the user along with the verification ot the computed moves. (Note, it is not trivial to create several automatic unit tests for this specific tasks.)
- Phase 2: Several randomly generated systems from the previous tests were used to play the game.

Additional features like modifications on the graph or take a Step-Back in the game have been only tested rarely on the fly.

4 Ongoing and Future work within T-Beg

4.1 Extending T-Beg

In the future we would like to extend our prototype implementation to an efficient coalgebraic partition refinement algorithm, adapting the ideas of Kanellakis/Smolka [4] or Paige/Tarjan [7, 2]. The latter method achieves runtime $\mathcal{O}(n \cdot \log n)$, where n is the size of the system, by using so-called *three-way-splitting* that chooses equivalence classes for splitting in a clever way.

We are also interested in studying applications where we can exploit the fact that the distinguishing formula witnesses non-bisimilarity. For instance, we see interesting uses in the area of differential privacy [1], for which we would need to generalize the theory to a quantitative setting. That is, we would like to construct distinguishing formulas in the setting of quantitative coalgebraic logics, which characterizes behavioural distances.

Furthermore, T-Beg can be extended to include a feature for the automatic generation of interesting game instances, more concretely in this context of non-bisimilar processes, which is a challenging problem as well.

Finally, we are interested in creating a self-explaining web interface which enables the use of the implemented functors. We also plan to extend T-Beg by additional functors (e.g. Mealy Machines, Finite Deterministic Automata).

4.2 Stabilizing T-Beg

We are planning to automatize the verification process of the winning strategy computation and the generation of the distinguishing formulas. Therefore, we want to integrate self-testing of the winning strategy computations by verifying automatically if a state satisfies such a generated formula.

References

1. Chatzikokolakis, K., Gebler, D., Palamidessi, C., Xu, L.: Generalized Bisimulation Metrics. In: Proc. of CONCUR '14. Springer (2014), LNCS/ARCoSS 8704
2. Dorsch, U., Milius, S., Schröder, L., Wißmann, T.: Efficient Coalgebraic Partition Refinement. In: Proc. of CONCUR 2017. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
3. de Frutos-Escrig, D., Keiren, J.J.A., Willemse, T.A.C.: Games for Bisimulations and Abstraction. CoRR **abs/1611.00401** (2016)
4. Kanellakis, P.C., Smolka, S.A.: CCS Expressions, Finite State Processes, and Three Problems of Equivalence. Inf. Comput. **86**, 43–68 (1990)
5. König, B., Mika-Michalski, C., Schröder, L.: Explaining Non-Bisimilarity in a Coalgebraic Approach: Games and Distinguishing Formulas. Website (2020), online available http://www.ti.inf.uni-due.de/fileadmin/public/tools/tbeg/tbeg.pdf; last access on 14th January.
6. König, B., Mika-Michalski, C.: (Metric) Bisimulation Games and Real-Valued Modal Logics for Coalgebras. In: Proc. of CONCUR '18. LIPIcs, vol. 118, pp. 37:1–37:17. Schloss Dagstuhl – Leibniz Center for Informatics (2018)
7. Paige, R., Tarjan, R.E.: Three Partition Refinement Algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)