DTEC: A new equilibria checker for omega-regular games

ABHEEK GHOSH, The University of Texas at Austin

We create an equilibria checker for finite multiplayer ω-regular games called Doomsday-Threatening Equilibria Checker (DTEC). In a doomsday equilibrium, players use a strategy profile where all players satisfy their own objective, and if any coalition of players deviates and violate a player's objective then the objective of every player is violated. [Chatterjee et al., 2014] introduced this equilibria concept and gave algorithms and complexity results for deciding its existence for various classes of ω-regular objectives.

DTEC has been implemented from scratch in C++. It provides a flexible and robust language to specify multiplayer graph games, comparable to popular model checking tools like NuSMV, Prism, Slugs, etc. It takes two input files, one that specifies the model, and the other gives the specifications. DTEC supports only reachability objectives, mainly because other, general, objectives have a bad time complexity as proved by [Chatterjee et al., 2014]. We also provide several examples that will help users create their own models and test them using the tool.

We also study several models for a four way traffic intersection. We observe that many reasonable models don't have a doomsday equilibrium, and finally refine them to get one that does have the equilibria. This project and the relevant papers we read introduced us to the area of finite multiplayer ω-regular games.

1 INTRODUCTION

Two-player games have been very popular and useful for reactive synthesis [Emerson and Jutla, 1991, Martin, 1975, Shapley, 1953]. In these games, it is usually assumed that one of the players, the system, plays with the objective to satisfy some given specifications. While the other player, which consists of all the other players or components in the environment, is assumed to be adversarial and wants to violate the specifications. But it has been observed that in many scenarios, like protocol synthesis, it is more realistic to assume that the players may have their own objectives, and may or may not be adversarial to each other. This has increased the recent interest in multiplayer games [Chatterjee et al., 2006, Fisman et al., 2010, Ummels and Wojtczak, 2011].

Nash equilibrium is the concept popularly used to capture rational behavior in multiplayer settings — the players are want to satisfy their own objective and are indifferent to other's objective [Nash, 1950]. In areas outside verification and synthesis, like economics and algorithmic game theory, it is fine to assume that the players are indifferent to the objectives of other player or even altruistic (Stackelberg games), and therefore Nash equilibrium and its modifications like Bayes-Nash equilibrium, correlated equilibrium, etc. are popularly used. But in synthesis, it may be more appropriate to assume adversarial external criteria, captured by concepts like secure equilibria, doomsday-threatening equilibria, etc. [Chatterjee et al., 2006, Ummels and Wojtczak, 2011].

2 DOOMSDAY-THREATENING EQUILIBRIA.

There are n players. The game is a tuple $(S, P, s_{\mathsf{init}}, \Sigma, \Delta, AP, L)$ such that

- S is a nonempty finite set of *states*,
- $P = \{S_1, S_2, \ldots, S_n\}$ is a partition of S into n classes of states, one for each Player $i \in \{1, 2, \ldots, n\}$,
- $s_{\mathsf{init}} \in S$ is the initial state,
- Σ is a finite set of alphabet,
- $\Delta : S \times \Sigma \to S$ is the transition function of the game,
- AP is the set of atomic propositions, and

- $L : S \longrightarrow 2^{AP}$ is the function that labels each state.

The game starts from an initial state s_{init} and then it goes on infinitely. Each player i has an LTL objective, say W_i. A player i has control over states in S_i and makes the move when the game is in those states. The strategy used by the player i to make the moves, using the previous history, when the game is in S_i is its strategy λ_i. Combining all the individual strategies, we get the strategy profile $\Lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n)$. When player i plays using strategy λ_i, and others can play arbitrarily, we get the set of paths outcome$_i(\lambda_i)$. And given a strategy profile we get a fixed path outcome$(\Lambda) = \bigcap$ outcome$_i(\lambda_i)$.

Doomsday-equilibrium. A strategy profile $\Lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n)$ is a doomsday equilibrium (DE) if:

(1) All players satisfy their objective, i.e. for all outcome$(\Lambda) \in \bigcap_i W_i$;
(2) and all players are able to retaliate in case of deviation: for all i, $1 \leq i \leq n$, for all path $\rho \in$ outcome$_i(\lambda_i)$, if $\rho \notin W_i$, then $\rho \in \bigcap_{i \neq j} \overline{W_j}$.

The paper studies different types of LTL specification. For each of the types, it gives algorithms with different time complexities.

(1) Reachability objectives. Some set of states are visited eventually. Ptime-complete.
(2) Safety objectives. Some set of states are never left. Pspace-complete, NP-hard. Ptime for a fixed number of players.
(3) Buchi objectives. Always eventually some set of states is visited. Ptime-complete.
(4) Co-Buchi objectives. Eventually always some set of states is visited. Ptime-complete.
(5) Parity objectives. Let $d \in \mathbb{N}$, a parity objective with d priorities is a condition which is defined with a parity function $p : S \longrightarrow \{0, 1, \ldots, d\}$ such that the minimal priority among the set of states that are visited infinitely often is even. Pspace, NP-hard, CoNP-hard.

For different types of objectives, different algorithms are given. For tail objectives — Buchi, co-Buchi, and parity — there is an algorithm which works with different time complexity for the different cases. There is a second different algorithm for reachability objective, and another for safety objectives. For general LTL specification, the original problem is converted to a new larger problem, and the algorithm given for the parity objectives is used.

3 DTEC

As mentioned before, DTEC has been implemented from scratch in C++. DTEC can be divided into two main components:

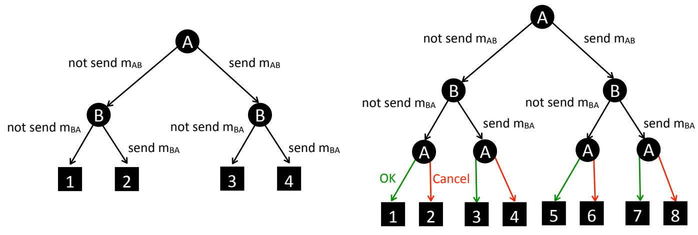
- The first part reads the .model and .specs files and constructs the model. DTEC supports a flexible and robust model specification format, similar to other tools like NuSMV, Prism, etc. This part of DTEC processes this model specification and sets up everything for the checker.
- The second part implements the algorithm 1 for reachability objective and checks whether a DE exists or not [Chatterjee et al., 2014]. If an equilibrium exists, it also outputs a path that makes everyone reach their objective.

The model file has seven sections corresponding to the game tuple $(S, P, s_{\text{init}}, \Sigma, \Delta, AP, L) \leftrightarrow$ (STATES, PARTITIONS, INITIAL STATE, ALPHABET, TRANSITION FUNCTION, ATOMIC PROPOSITIONS, LABELING FUNCTION). It has features like arrays, including negative range based indices, good lexicographic analysis, parsing, and error checking. The code has been submitted to Canvas, and it is also available at Github: https://github.com/AbheekG/dtec with usage instructions. The

Algorithm 1 Equilibrium checking algorithm for reachability objectives.

```
Require: Model, Reachability set for each player $T_i$
1: for all player $i$ do
2:     Find the set of states $F_i$, from which player $i$ has a strategy to force the game to eventually
       reach $T_i$ against all other players. (2-player reachability game)
3:     Find the set of states $R_i$, from which player $i$ has a strategy to always force doomsday $(\cap_j \overline{T_j})$
       or $F_i$. (2-player safety game)
4: end for
5: Find a path that starts from $s_{\text{init}}$ and stays in $\cap_i R_i$ until it reaches a state $s \in S$ such that $s \in T_i$
   for some $i$ and $s \in \cap_j F_j$.
6: return If such a path exists, return true, else return false.
```

Fig. 1. Fair-Exchange Protocols 1 (4.1) and 2 (4.2)



example *hello_world* (*hello_world.model* and *hello_world.specs*) explains the usage along with a running example.

4 EXAMPLES

In this section, we provide some examples, most of them based on the traffic intersection.

Example 4.1. [Chatterjee et al., 2014] Fair-Exchange Protocol 1 (first example given in the paper) [Files: *paper_1.model*, *paper_1.specs*]. There are two players: *A* and *B*. They are exchanging items, like money or property. First, *A* decides whether to give or not give (say property), then *B* decides to give or not give (say money). They both want to receive the item (reachability specification for such a state) and would prefer not giving the item if possible.

Example 4.2. [Chatterjee et al., 2014] Fair-Exchange Protocol 2 (second example given in the paper) [Files: *paper_1.model,paper_1.specs*]. This example is similar to 4.1, except that in this there is a third step where *A* can cancel the deal and all transactions.

There are two players: *A* and *B*. They are exchanging items, like money or property. First, *A* decides whether to give or not give (say property), then *B* decides to give or not give (say money), then *A* executes a veto to go forward or cancel the deal. They both want to receive the item (reachability specification for such a state) and would prefer not giving their item if possible.

In Example 4.1, there doesn't exist any DE. Whether *A* gives the property or not, *B* has a dominating strategy to not give the money, which leaves *A* with no option other than to not give the property. But in Example 4.2, there exists a DE. In the first step, *A* decides to give the property, risk-free because it has the option to cancel the transaction later, and if it doesn't give the property then *B* will also not give the money. If *A* gives the property, then *B* will give the money because otherwise, it knows that A will cancel the transaction. And in the last step, A doesn't cancel the transaction because doing that will cancel the transfer of both money and property.

Example 4.3. Traffic Protocol 1 (No DE) [Files: $traffic\_1.py$ (generates the .model file), $traffic\_1.model$, $traffic.specs$]. Four-way intersection. Four types of cars. Only one can go at a time, if more than one goes at the same time then they collide.

Decide order: 1 - 2 - 3 - 4 - EXEC. This keeps on repeating infinitely.

Each car can decide to:

- W: Wait
- G: Go

Each car can be in one of these four states:

- W: Waiting
- G: Go
- R: Reached
- C: Crashed

Initially, everyone is in W (Waiting) state. While in W, if only one car decides G (Go), it reaches R (Reached). If more than one cars decide G (Go), the cars that decided to G (Go) reaches C (Crashed). Once a car is in R or C, it stays there. We assume that if some cars crash in the current time step (iteration), it doesn't affect the next time step and other cars can possibly reach successfully then.

The combined state is a tuple of five components. The first four components correspond to individual states of the four cars. And the last component specifies who makes the current move (who controls the state) and it ranges from 0 to 4. 1-4: the four players. 0: for the execution by the system depending upon previous moves (deterministic, no uncertainty). In our game there is no system, but we can assign the states belonging to the system to any arbitrary player as it doesn't create any problem because, as mentioned, the next state is fixed for any action.

The Traffic Protocol 1 (Example 4.3) doesn't have any DE. We present modified protocols in Examples 4.4 to 4.10. Examples 4.3 to 4.8 don't have a DE, but Examples 4.9 to 4.10 have a DE.

Example 4.4. Traffic Protocol 2 (No DE) [Files: $traffic\_2.py$ (generates the .model file), $traffic\_2.model$, $traffic.specs$]. The model is similar to Example 4.3, except now we give the cars multiple chances before each execution, and therefore more memory. We tried this to allow cars to collaborate, but doesn't help.

Order: 1 - 2 - 3 - 4 - 1 - 2 - 3 - 4 - ... - 4 - EXEC - (repeat)

Example 4.5. Traffic Protocol 3 (No DE) [Files: $traffic\_3.py$ (generates the .model file), $traffic\_3.model$, $traffic.specs$]. Modification to 4.4, here when more than one cars G (Go), then they don't C (Crash) but again reset to W (Waiting).

Example 4.6. Traffic Protocol 4 (No DE) [Files: $traffic\_4.py$ (generates the .model file), $traffic\_4.model$, $traffic.specs$]. Modification to 4.4. As explained in 4.4, the cars have multiple time points to decide their move before each execution and they could only take action from W (to W or G), now they are allowed to go from G to W also.

Example 4.7. Traffic Protocol 5 (No DE) [Files: $traffic\_5.py$ (generates the .model file), $traffic\_5.model$, $traffic.specs$]. As we are finding it difficult to get a model with a DE, we go back to Traffic Protocol 1 (4.3) and further make it easier to find a DE. We allow car 1 and car 3 to pass simultaneously without crashing, same for car 2 and car 4. It follows from the real-life observation that the opposite ways (like North-South, East-West) can go straight simultaneously.

Example 4.8. Traffic Protocol 6 (No DE) [Files: $traffic\_6.py$ (generates the .model file), $traffic\_6.model$, $traffic.specs$]. The model is similar to Traffic Protocol 2 (Example 4.4), where we gave the cars multiple chances before each execution, but now we also add a booking round after each iteration. So the order of steps looks like:

1 - 2 - 3 - 4 - BOOK - 1 - 2 - 3 - 4 - BOOK - ... - 4 - BOOK - EXEC - (repeat)

There is also a state B (Booked) for the cars, in addition to the previous four: W, G, R, and C. If there is only one G (Go) in a given iteration, then the booking step (BOOK) makes the car's state B (Booked), otherwise if multiple cars select G then they all reach C (Crashed). In the final execution step (EXEC), all the B (Booked) cars reach R (Reached).

Example 4.9. Traffic Protocol 7 (DE exists!) [Files: $traffic\_7.py$ (generates the .model file), $traffic\_7.model$, $traffic.specs$]. The model has all the components of Traffic Protocol 6 (Example 4.8), like multiple steps before execution (EXEC) and booking (BOOK), additionally, this new model has a Veto iteration before the EXEC. Now the order of steps look like:

1 - 2 - 3 - 4 - BOOK - 1 - 2 - 3 - 4 - BOOK - ... - 4 - BOOK - Veto1 - Veto2 - Veto3 - Veto4 - EXEC - (repeat)

In the Veto iteration, a car can cancel all bookings, i.e., all B (Booked) cars go back to W (Waiting). This allows a car to create doomsday if it doesn't get booked. If the number of booking iterations is greater than or equal to the number of cars, then they reach a DE.

Example 4.10. Traffic Protocol 8 and 9 (DE exists!) [Files: $traffic\_8.py$ and $traffic\_9.py$ (generates the .model file), (the model is not attached due to large size ($\tilde{1}$ GB) but can be generated using the .py file), $traffic_8.specs$ and $traffic_12.specs$]. This is similar to Traffic Protocol 7 (Example 4.9) with the additional feature of selective crashing as introduced in Example 4.7 and twelve types of cars going:

$$\{East, West, North, South\}x\{Left, Right, Straight\}$$

. In

For example, cars going East-Straight and West-Straight don't crash, but cars going East-Stright and West-Left crash. In these examples, if the number of booking iterations is greater than or equal to four (less than types of cars), then they reach a DE.

5 CONCLUSION

In the examples, we saw the types of models that can have a DE and also the ones that don't. In the first six traffic examples, it can be observed that for the car 1, there was no move that could guarantee R (Reached) for itself or C (Crashed) for all. If the car 1 decided G (Go) then it could be crashed by some other car and they both could reach C, while some other car can reach R in some other iteration. But in the last few examples, by giving the veto power to the cars/players and having a sufficient number of booking rounds, all cars can book a slot and successfully reach their destination.

The DTEC has an easy and robust input language and can be used for checking DE with reachability objectives by researchers in the area. We also learned about multiplayer games and their relation with formal methods and reactive synthesis.

REFERENCES

Krishnendu Chatterjee, Laurent Doyen, Emmanuel Filiot, and Jean-François Raskin. 2014. Doomsday equilibria for omega-regular games. In *International Conference on Verification, Model Checking, and Abstract Interpretation.* Springer, 78–97.

Krishnendu Chatterjee, Thomas A. Henzinger, and Marcin Jurdziflski. 2006. Games with secure equilibria. *Theoretical Computer Science* 365, 1 (2006), 67 – 82. https://doi.org/10.1016/j.tcs.2006.07.032 Formal Methods for Components and Objects.

E Allen Emerson and Charanjit S Jutla. 1991. Tree automata, mu-calculus and determinacy. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*. IEEE, 368–377.

Dana Fisman, Orna Kupferman, and Yoad Lustig. 2010. Rational Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*, Javier Esparza and Rupak Majumdar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–204.

Donald A. Martin. 1975. Borel Determinacy. *Annals of Mathematics* 102, 2 (1975), 363–371. http://www.jstor.org/stable/1971035

John F. Nash. 1950. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences* 36, 1 (1950), 48–49. https://doi.org/10.1073/pnas.36.1.48 arXiv:https://www.pnas.org/content/36/1/48.full.pdf

Lloyd S Shapley. 1953. Stochastic games. *Proceedings of the national academy of sciences* 39, 10 (1953), 1095–1100.

Michael Ummels and Dominik Wojtczak. 2011. The complexity of Nash equilibria in stochastic multiplayer games. *arXiv preprint arXiv:1109.4017* (2011).