Python Implementation of ATL

Chris Tighe, Matt Sehgal
Brandeis University
Waltham, Mass.
ctighe@brandeis.edu, matt1913@brandeis.edu

Abstract
This project seeks to implement Alternating-time Temporal Logic (ATL) in Python. ATL is a branching-time temporal logic for evaluating if a system is able to resolve its internal choices so that the satisfaction of a property is guaranteed no matter how the environment resolves the external choices. This implementation will provide a two player turn-based ATL model checking system in Python as a base. This base will lay the foundation to continue to explore what is possible in multi-agent scenarios with various turn structures.

Keywords: ATL, Python, Temporal Logic

1. Introduction

The purpose of this project is to implement Alternating-time Temporal Logic (ATL) in Python. ATL is a branching-time temporal logic that is used to evaluate whether a system is able to guarantee the satisfaction of a property, regardless of how the external choices made by the environment are resolved. ATL model checking is used to formally verify the behavior of systems over time, in order to ensure that certain properties or specifications are satisfied. It is particularly useful in cases where the behavior of a system depends on the choices made by multiple agents, or where the system's behavior changes over time. ATL allows for the analysis of such systems by representing them as models and using logical expressions to represent properties or specifications that need to be satisfied. Some common applications of ATL model checking include the verification of software systems, the analysis of communication protocols, and the evaluation of security protocols. This implementation will provide a base for a two player turn-based ATL model checking system in Python, which will serve as a foundation for further exploration of multi-agent scenarios with various turn structures. The implementation will be carried out primarily through the lens of the Train-Gate problem and Bloisi's examples with it.

2. Background and Related Work

While linear-time and branching-time logics are natural specification languages for closed systems, ATL is a formalism that allows for the specification and verification of properties of open systems. ATL offers selective quantification over those paths that are possible outcomes of games, such as the game in which the system and the environment alternate moves. While linear-time and branching-time logics are natural specification languages for closed systems, alternating-time logics are natural specification languages for open systems. The problems of receptiveness, realizability, and controllability can be formulated as model-checking problems for alternating-time formulas. This makes it particularly useful for the specification of properties of systems that interact with their environment, such as communication protocols or reactive systems.

There have been several papers that have contributed to the development and understanding of ATL model checking. Belardinelli et al. presented semantics for ATL, both for imperfect and perfect recall, as well as a system of optimal algorithms for model checking in ATL (2018). Vester provided implementations and proofs for the decidability of certain implementations of ATL (2013). Bloisi's presentation was also helpful in providing definitions and starting points for ATL implementations (2008). Finally, this project heavily referred to the work done by Alur et al. which carefully explains the specifications of ATL and model checking in ATL (2002). The work referenced from Bloisi draws from Alur et al.'s paper. Our project aims to implement ATL in Python, building upon the work of these previous researchers. We will be applying some of the analogous theory presented in Belardinelli et al.'s paper to our implementation, while also considering the decidability results provided by Vester and the definitions and starting points provided by Bloisi and Alur et al.

3. Methods

Alternating-time Temporal Logic (ATL) is implemented in the code through 4 files. In the first file, States.py, states are implemented by the **State** class which contains attributes for the *player* (if any), *props* (propositions), *connections* (to other states), and *color* (for graph traversal purposes). In the next file, Expressions.py, logical expressions are implemented in the **Exp** class which contains attributes for the potential subexpressions, *subexp1* (existence is guaranteed), *subexp2*, and the operator, *op*. **Exp** also contains the method *check(state)* which checks if an expression is satisfied in the model. In the next file, util.py, parsing is implemented for logical expressions in string form by the *parse(text)* method. ATL expressions can be represented as strings according to Table 1.

Constant	"true" or "false"
Proposition	any string of characters
Negation	`~`
Disjunction	`V`
Conjunction	`^`
Implication	`->`
Players	`{}, {_}, or {_,...}`
Circle	`@`
Diamond	`<>`
Square	`[]`
Until	`U`

Table 1: Symbols for ATL expressions as strings

These expressions are then parsed into a final encompassing **Exp** instance consisting of all subexpressions. For example, in the Train-Gate Problem shown in Figure 1:

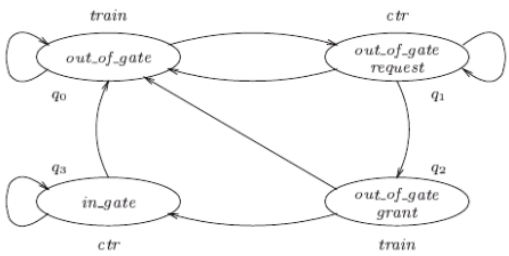

Figure 1: Train-Gate Problem state diagram

the expression for "whenever the train is out of the gate and does not have a grant to enter the gate, the controller can prevent it from entering the gate" can be represented formally as:

⟪∅⟫□((out_of_gate ∧ ¬grant) → ⟪ctr⟫□(out_of_gate))

and converted to a string representation as:

"{0}[]((out_of_gate ^ ~grant) -> {ctr}[](out_of_gate))"

In the final file, ATL.py, there are two methods, *test(exp)* and *is_valid(exp, states)* which facilitate the model checking. The method *test(exp)* is used to test if an expression, *exp*, is satisfied in all states of the system. The method *is_valid(exp*) is used to check if an expression, *exp*, is valid in all given states, *states*. Through these two methods model checking can be performed as systems can be constructed from **State** instances, expressions can be created as strings and parsed into **Exp** instances which can then be checked for satisfaction or validity against the system.

4. Results

4.1 Validity Testing

Some example valid expressions for one ATL system we used (as shown in Fig. 1) were also provided by Bloisi. These were extensively checked, and all were confirmed to be valid under our testing.

4.2 Satisfiability Testing

Under similar circumstances we designed some expressions that were intended to evaluate to true only sometimes. This involved some extra work since many real-world examples of ATL expressions begin with the player-temporal quantifier ⟪∅⟫□, which represents quantification over all possible paths irrespective of any player control.

Testing with basic first-order logic expressions, their satisfiability was confirmed and matched hand-computed results. Further testing with quantified expressions confirms that they are still evaluated accurately.

5. Discussion

5.1 Assumptions and Limitations

Our ATL implementation had some assumptions made to simplify the process. The biggest assumption was that we would be implementing a turn-based ATL system- this means that each state of the transition graph is "controlled" by exactly one player, so the simulation's complexity can be dramatically reduced. This is not the only form of ATL that exists; there are also Moore-synchronous ATL systems in which every player makes a concurrent choice at each step, and the combination of choices is what determines the state transition. This assumption is a significant limitation to the world of ATL systems that can be simulated, however it can be mitigated by transforming the Moore-synchronous system into a turn-based system through a process described in Appendix A. Briefly, each original state is broken down into a subtree that is bounded by additional propositions to allow for complex expressions in the source system to be translated into the target system.

Another limitation of our system is that our expression-handling code is designed in such a way that it cannot effectively handle dual formulae in an efficient manner. If we wanted to express a statement that says a group of players cannot cooperate to avoid making a subexpression true, we would have to express that in a complex system of nested expressions instead of the simpler expression Bloisi lists on slide 24. This is because our code can't handle different types of player-scope quantifiers, only path quantifiers.

5.2 Potential Extensions

Our implementation is not fully complete. It certainly contains enough functionality to test the validity of ATL statements, however there are some additional features that could be implemented to make it more generally useful. We left the implementation of the proof in Appendix A as an exercise for the reader, but in addition we didn't address possible dual modalities of player or path quantifiers. These could be included to ease the burden on any potential user.

We also didn't implement any kind of user interface for constructing a set of states on which to evaluate new expressions; this is because the current method of creating states is far less clunky than our original method of constructing a nesting tree structure of expressions, and would only need to be performed once to test arbitrarily many expressions.

6. Conclusion

In conclusion, our implementation of Alternating-time Temporal Logic (ATL) in Python has been successful in modeling the Train-Gate problem and satisfying the desired properties. As previously mentioned, ATL is a useful formalism for the specification and verification of properties of open systems, such as communication protocols, reactive systems, and software systems. Our implementation has provided a two player turn-based model checking system that can be built upon in the future to explore multi-agent scenarios with various turn structures.

In appendix A, we provide a proof that demonstrates the soundness and completeness of our implementation of Moore-synchronous ATL systems using turn-based systems. The proof outlines the process of converting an original ATL graph into a turn-based system by creating a tree rooted at the number of players with control at each state, with each layer having nodes representing the number of choices each player has at that state. New states in the tree gain the propositions of the tree's root, and each leaf is connected to the original target state via recorded outgoing edges, gaining a new unique proposition in the process. The proof also explains how ATL statements in the original system must be modified to preserve their

validity in the reduced form, including the translation of square, until, and circle temporal operator statements. This proof demonstrates the validity of our implementation in accurately representing and evaluating the satisfaction of properties in the model.

Our implementation of ATL in Python has proven to be a useful tool for the analysis of open systems and the formal verification of properties over time. We believe that this implementation lays a strong foundation to continue work towards model checking of more complex multi-agent systems and provides a sound and easy way to experiment with ATL.

7. Bibliographical References

Belardinelli, F., Lomuscio, A., Murano, A., & Rubin, S. (2018, July 13). Alternating-time temporal logic on finite traces. IJCAI. Retrieved December 16, 2022, from https://www.ijcai.org/Proceedings/2018/11

Vester, S. (2013, July 17). *Alternating-time temporal logic with finite-memory strategies*. arXiv.org. Retrieved December 17, 2022, from https://arxiv.org/abs/1307.4476

Alur, R., Henzinger, T. A., & Kupferman, O. (2002). Alternating-time temporal logic. Alternating-time Temporal Logic. Retrieved December 19, 2022, from https://www.cis.upenn.edu/~alur/Jacm02.pdf

Bloisi, D. (2008, November 20). *An introduction to alternating-time temporal logic*. An introduction to ATL. Retrieved December 17, 2022, from https://www.diag.uniroma1.it/~bloisi/didattica/apprendimento0809/atl.pdf

Appendix A:

Proof that Moore-synchronous ATL systems can be represented with turn-based systems:

For every state q in the original ATL graph:

Record all outgoing edges (chosen states of each player, which state they connect to), preserve all incoming edges

Create a n-level tree rooted at q ($n =$ number of players with control at state q)

Each layer $l > 0$ has $m * l - 1$ nodes, where $m =$ the number of choices that player l has in q.

Each new state in the tree gains the propositions in the tree's root

The root of the tree gains a new unique proposition, called $substate\_start$

Each leaf of the tree is connected via an outgoing edge from the recorded edges to the original target state (or the

root of that substate tree) and gains a new unique proposition called $substate\_end$

In this way, each state of the original tree is represented using an asynchronous turn-based state transition subgraph while preserving the overall transition flow between individual "superstates" of the original ATL graph. All that remains is to detail how ATL statements for the original system need to be modified to preserve validity in the reduced form:

Square statements such as $\langle\langle A\rangle\rangle\Box\ \varphi$ become $(substate\_start \rightarrow \langle\langle A\rangle\rangle\Box\ \varphi)$

Until statements such as $\langle\langle A\rangle\rangle\varphi_1 \cup\ \varphi_2$ become $(substate\_start \rightarrow \langle\langle A\rangle\rangle\varphi_1 \cup\ \varphi_2)$

Circle statements such as $\langle\langle A\rangle\rangle \bigcirc \varphi$ become
$(substate\_start \rightarrow \langle\langle A\rangle\rangle(\neg substate\_end)\ \cup\ (substate\_end \wedge \langle\langle A\rangle\rangle \bigcirc \varphi))$

Square and Until statements describe nonspecific-length paths, so their translated versions are relatively simple- modified slightly to make sure that their validity is only tested at substates that haven't had any "substate choices" made yet.

Circle statements are more complex since they are intended to only check immediate neighbors, but that relation has been disassociated by the addition of so many new nodes in the graph. The relation is reconstructed through the use of the extra propositions to identify the boundaries of each original state's substate tree.