A Statistical Model Checker for Situation Calculus Based Multi-Agent Models

(Extended Abstract)

Christian Kroiß
Institute for Informatics
Ludwig-Maximilians-Universität München
Munich, Germany
kroiss@pst.ifi.lmu.de

ABSTRACT

In this paper we introduce a new approach for multi-agent simulation and statistical model checking that combines the well-established situation calculus with a first order version of bounded linear time logic (BLTL). This creates a fully integrated solution for specifying system behavior and requirements within the same logical framework. We realized the approach in an extensible tool that combines the benefits of constraint logic programming with the versatility of Python and its ecosystem. First experiments show that the approach is applicable to a wide range of problems and that altogether a more flexible modeling-verification workflow is achieved than in most existing solutions.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.1.6 [**Programming Techniques**]: Logic Programming

Keywords

Statistical Model Checking; Agent-Based Simulation; Multi-Agent Systems; Situation Calculus

1. INTRODUCTION

Statistical model checking (SMC) [1] can be seen as a variant of simulation where statistical hypothesis tests are applied to recorded execution traces. The goal is to assure that certain properties are violated only with a given maximal probability. The well-known state-space explosion problem of exact model checking is avoided in SMC while still allowing to specify the evaluated properties in variants of temporal logics, e.g. LTL [2]. However, in existing SMC solutions there is usually a trade-off between expressiveness of the property specification language and the versatility of the system model: either the SMC engine works on traces recorded from a black-box system, in which case a problem-specific interpretation of the traces has to be specified; or the system model is defined in a modeling language that provides direct access to the simulated system's state. In the latter case, however, the nature of the used simulation engine often enforces unwanted abstractions, e.g. due to lacking support for more complex data processing facilities.

In order to achieve a more direct integration, we propose to base the system model on the situation calculus [3], a well-known first-order logic language for describing dynamic systems. We combine this with a specialized first-order version of LTL that allows direct reasoning about all facets of the system state. The approach is realized by a Python-based simulation engine that connects to the ECL^{i}PS^e constraint logic programming environment [4] for property evaluation and for calculating system state updates. Agent control processes, as well as probability distributions for stochastic actions and events, are defined using an extensible Python framework. Thus, the modeler can profit from the advantages of a consistent logical model and at the same time use the full Python ecosystem, e.g. for advanced mathematical. An overview about this architecture is given in Figure 1.

Appears in: *Alessio Lomuscio, Paul Scerri, Ana Bazzan, and Michael Huhns (eds.), Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), May 5-9, 2014, Paris, France.*

2. THE MODELING FRAMEWORK

Every simulation model in our approach is at its core based on a situation calculus axiomatization of the agents and their environment. Concretely, the model contains a *precondition axiom* for each *action*, as well as a *successor state axiom (SSA)* for each *fluent* (i.e. situation-dependent feature). Each SSA defines an update rule for the fluent's value as a function of the current *situation* and the performed action. In practice, all axioms are defined using the ECL^{i}PS^e Prolog dialect. However, we present an example of a SSA in a more compact mathematical notation:

$$holding(r,i,do(a,s)) \equiv a = grab(r,i) \vee (holding(r,i,s) \wedge a \neq drop(r,i))$$

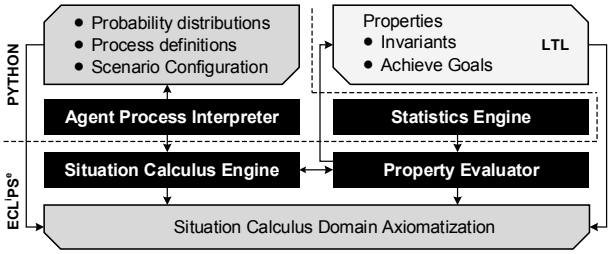

Figure 1: Tool Architecture

This axiom simply says that a robot r is holding an item after it has grabbed it and that it keeps holding it until the item is dropped. The last argument $do(a, s)$ on the left side denotes the *situation* that is reached after the action a is performed in situation s.

Given an axiomatization along these lines, the simulation engine uses *progression* [3, ch. 9] to calculate the new world state after executing actions yielded by the agents in each simulation step. In this model, time is represented as a regular fluent (*time*) that is increased by an action (*tick*) which is implicitly performed at the end of each step. Of course, the modeler has to specify *control procedures* for all agents in the system. This is done with a Python API that provides all common procedural constructs like `while`, `if`, variable assignments, or procedure calls. Within the procedures the current values of all fluents are accessible, and calculations may be performed both by Python and ECL^{i}PS^e (Prolog) functions. A simple agent procedure might look like this:

```
1 Procedure("carryAway",[], [
2   Select("reachable",[SELF, ("i","item")]),
3   Act("grab", [SELF, Var("i")]),
4   While("xpos(self)␣<␣100", [],
5     Act("move_right", [SELF])
6   ), Act("drop", [SELF, Var("i")])])
```

The procedure `carryAway` first assigns a value for variable i by selecting an arbitrary item that makes the Prolog predicate `reachable(SELF, i)` true. Then the action `grab` is executed, followed by a loop that executes `move_right` until the Python expression `xpos(self) < 100` becomes false. Once the robot has reached its target, it drops the item.

With the situation calculus model at hands, one can define properties using a first-order logic version of bounded LTL (i.e. LTL with upper time bounds attached to the temporal operators). An example in the context of the robot scenario from above could be the following formula (like before using a mathematical notation instead of the concrete syntax):

$$\forall r \in R. \forall i \in I.\, occur(grab(r, i)) \rightarrow \\ holding(r, i)\, \mathsf{U}^{50} (xpos(r) \geq 100 \ \wedge \\ \neg(\exists r' \in R.\, r' \neq r \wedge xpos(r') \geq 100))$$

Here R and I denote the domains of robots and items, respectively, and U^{50} is the LTL "until" operator, bounded to a limit of 50 time steps. Hence, the property asserts that whenever a robot r grabs an item i, r will eventually arrive at a region defined by $x \geq 100$ within at most 50 time steps. It also requires that when r arrives, there will not be another robot in the target region, and until r arrives it keeps holding the item. Even in this simple formula, it becomes obvious how useful the first-order variant of LTL is in the context of multi-agent systems. In particular, it allows to explicitly distinguish individual agents and entities and to reason about relations between them.

Since we restrict the system to finite domains for each entity sort (like robot or item) and all temporal operators are bounded, our property evaluator is in any case able to verify the satisfaction of formulas like the one above in finite time during simulation. If the model contains stochastic actions or events then the number of property violations becomes a binomially distributed random variable. One can now apply well-known statistical methods, either to estimate the probability p of a property violation or to test an hypothesis like $H_0 : p \leq p_0$. For the latter task, we integrated the sequential probability ratio test (SPRT) by A. Wald [5]. This test allows performing a hypothesis test without a predefined sample length. Instead the engine decides after each experiment (i.e. simulation run) whether enough data has been collected or if additional simulation runs are necessary to satisfy the chosen error probability bounds.

3. EVALUATION

The tool behind our approach is available at [6]. So far we have been working with several smaller models that were designed specifically for testing the tool and for exploring the expressiveness of the modeling languages. Besides that, we are currently working on a case study of the EU project ASCENS. In the main scenario, which is described in [7], parking-/charging places for electric vehicles are assigned using a distributed optimization scheme. We were able to create a simulation model that contains a relatively detailed representation of the vehicles' navigation on a graph-based map and also of the message-based communication protocols between vehicles and control stations. Here the value of the flexible language integration became very obvious. Among others, it permitted us to leverage Prolog's list processing and term matching capabilities to model message passing while at the same time using a Python graph library for route calculation. Furthermore, the fact that basically all features of the system are directly accessible in property formulas notably facilitates an agile iterative modeling process. In the next step our approach will be applied to analyze the model with respect to aspects like communication failure resilience and adaptivity. Due to space restrictions, we are not able to present further results here. However, the experiences so far are very promising and we plan to publish a detailed treatment soon.

4. ACKNOWLEDGEMENTS

This work has been partially sponsored by the EU project ASCENS, FP7 257414.

5. REFERENCES

[1] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Runtime Verification*, pp. 122–135, Springer, 2010.

[2] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science*, SFCS '77, (Washington, DC, USA), pp. 46–57, IEEE Computer Society, 1977.

[3] R. Reiter, *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT press, 2001.

[4] "The ECLiPSe Constraint Programming System." http://www.eclipseclp.org.

[5] A. Wald *et al.*, "Sequential tests of statistical hypotheses," *Annals of Mathematical Statistics*, vol. 16, no. 2, pp. 117–186, 1945.

[6] http://www.salmatoolkit.org.

[7] T. Bures *et al.*, "A life cycle for the development of autonomic systems: The e-mobility showcase," in *Proceedings of the 3rd Workshop on Challenges for Achieving Self-Awareness in Autonomic Systems, Philadelphia, USA*, 2013.