soid: A Tool for Legal Accountability for Automated Decision Making

Samuel Judson[1(✉)], Matthew Elacqua[1], Filip Cano[2], Timos Antonopoulos[1], Bettina Könighofer[2], Scott J. Shapiro[3], and Ruzica Piskac[1]

[1] Yale University, New Haven, USA
{samuel.judson,matt.elacqua,
timos.antonopoulos,ruzica.piskac}@yale.edu
[2] Graz University of Technology, Graz, Austria
{filip.cano,bettina.koenighofer}@iaik.tugraz.at
[3] Yale Law School and Yale University, New Haven, USA
scott.shapiro@yale.edu

Abstract. We present soid, a tool for interrogating the decision making of autonomous agents using SMT-based automated reasoning. Relying on the Z3 SMT solver and KLEE symbolic execution engine, soid allows investigators to receive rigorously proven answers to factual and counterfactual queries about agent behavior, enabling effective legal and engineering accountability for harmful or otherwise incorrect decisions. We evaluate soid qualitatively and quantitatively on a pair of examples, i) a buggy implementation of a classic decision tree inference benchmark from the explainable AI (XAI) literature; and ii) a car crash in a simulated physics environment. For the latter, we also contribute the soid-gui, a domain-specific, web-based example interface for legal and other practitioners to specify factual and counterfactual queries without requiring sophisticated programming or formal methods expertise.

1 Introduction

Recent advances in (often ML-based) artificial intelligence have led to a proliferation of algorithmic decision making (ADM) agents. The risk that these agents may cause harm – and the many demonstrated examples of them already doing so, ranging across numerous domains [3,8,19,30] – has led to a significant demand for technologies to enable their responsible use. In this work, we present soid, a tool based on Judson *et al.*'s method [16] to account for software systems using computational tools from the fields of formal methods and automated reasoning. The soid tool is primarily oriented towards supporting legal reasoning and analysis, in order to better understand the ultimate *purpose* of an agent's decision making – as is often relied upon by various bodies of law.

In particular, rather than traditional verification methods which aim towards proving a specific program property, soid instead aims to 'put the agent on the stand'. The design of soid enables factual and counterfactual querying – underlying a finding of fact – in support of human-centered assessment of the 'why' of the agent's decision making. Such an assessment can then in turn justify holding responsible an answerable owner or operator, like a person or company. We

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14682, pp. 233–246, 2024.
https://doi.org/10.1007/978-3-031-65630-9_12

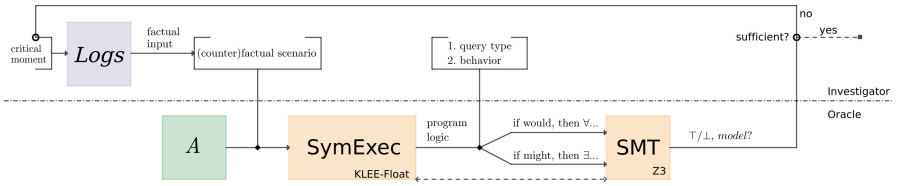


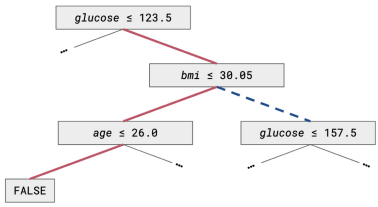
Fig. 1. Architecture of the soid tool.

describe the functioning of the soid tool itself as well as a pair of examples of its use on simulated harms. We also describe the soid-gui, a domain-specific interface for soid applied to autonomous vehicles, allowing for adaptive and interpretable analysis of driving decisions without requiring extensive programming skills or familiarity with formal logical reasoning.

The basic flow of soid, depicted in Fig. 1, is adaptive and requires a human in the loop. The human investigator – likely a practitioner such as a lawyer or regulator supported as necessary by engineers – uses soid to better understand the decision making of an agent program A. They do so by finding critical decision moments in the logs of A that transpired in the lead up to a harm, and then relaxing or perturbing the program inputs to specify a (family of) counterfactual scenario(s). The investigator then formulates a query asking what the behavior of A 'might' or 'would' have been [20] under that (family of) counterfactual(s). As we show in the design of our soid-gui, such questions can even be formulated in user-friendly interfaces that abstract away all of the formal logic and reasoning of soid for non-technical practitioners. Once a query is posed, a verification oracle using SMT-based automated reasoning – including constrained symbolic execution – gets the investigator a prompt answer. They can then continue to 'interrogate the witness' until they are satisfied they have a sufficient understanding of the purpose of A's decisions, and terminate the loop.

Contribution. In summary, we developed a command line tool and Python library soid, which uses symbolic execution (through Z3) and SMT solving (through KLEE) to enable rigorous interpretation of the decision-making logic of an autonomous agent. We demonstrate soid on a pair on instructive involving machine-learned agents. In both cases, we find soid able to resolve counterfactual queries with reasonable efficiency, even when adaptively posed through the interpretable soid-gui aimed at non-technical practitioners.

A Motivating Example. Consider a program A which computes a decision tree in order to classify the diabetes health risk status of an individual, a classic example in automated counterfactuals with legal implications due to [31]. The decision tree and code of A are shown in Fig. 2. However, the software system surrounding A creates an implicit unit conversion bug: A computes the body-mass-index (BMI) input to the decision tree, using height and weight parameters from its input. But, A expects metric inputs in kg and m and so computes the BMI without a necessary unit conversion, while the program inputs are instead




```
1  int traverse(Node *N, double *fv) {
2    if (N->class >= 0) return N->class;
3    return (fv[N->tidx] <= N->test)
4      ? traverse(N->tchild, fv)
5      : traverse(N->fchild, fv);
6  }
7
8  int classify(Node *root, double *data) {
9    double bmi = data[6] / pow(data[5], 2);
10   double fv[8] = {
11     data[0], data[1], data[2], data[3],
12     data[4],      bmi, data[7], data[8]
13   };
14
15   return traverse(root, fv);
16 }
```

Fig. 2. An incorrect decision tree classification. At left the decision subtree with the incorrect path in bolded red and the missed 'correct' branch in dashed blue. At right, the decision tree inference logic as implemented in C. (Color figure online)

provided in the imperial *in* and *lb*. Notably, A is 'correct' with respect to natural specifications – as is the decision tree in isolation. The flaw occurs due to a mistake in the composition of the software system as a whole. Nonetheless, the system misclassifies many inputs, as $(kg/m^2) \gg (lb/in^2)$ for the same quantities.

The goal of soid is to enable a legal practitioner to understand the presence of and conditions underlying a potential misclassification. Unlike statistical methods for counterfactual analysis which only analyze the (correct) decision model [31], the minimal assumptions underlying soid – namely, the lack of an assumption that the broader software system correctly uses the decision model – make it a more capable framework for analyzing this type of 'implicit conversion' failure. In §2.1 we run a small empirical analysis on A, showing how soid enables a user to specify concrete factual and counterfactual queries to understand the conditions under which the failure can occur and their implications.

1.1 Related Work

The explainable AI (XAI) and fairness, accountability, and transparency (FAccT) communities have developed numerous methods and tools for enabling accountability of ADMs, machine-learned or otherwise, for which [1,10,13] are recent surveys. The closest tool to soid of which we are aware is the VerifAI project [9,11]. Many of these tools and techniques focus on counterfactual reasoning in particular [7,14,15,24,31]. In comparison to the prevailing lines of this research, soid emphasizes i) after-the-fact (or *ex post*) analysis for algorithmic accountability in the style of with legal reasoning; ii) the use of SMT-based verification technologies capable of resolving counterfactual questions about whole families of scenarios; and iii) emphasis on the 'code as run', rather than evaluating a specific component like a particular decision model, or requiring an abstracted program representation or a formal model of the (often complex social and/or physical) environment the agent operates within.

2 soid Tool Architecture and Usage

Figure 1 illustrates the architecture of soid. The tool is implemented in Python, and invokes the Z3 SMT solver [26] for resolving queries.

```
@soid.register
def environmental(E):
  return And(And(Equal(E.occupied_0_0, False)),
             ...     # omitted for brevity
             And(Equal(E.occupied_2_1, True), Equal(E.orient_2_1, cardinals['East'])),
             ... )   # omitted for brevity

@soid.register
def state(S):
    return And(Equal(S.curr_direction, cardinals['North']),
               Equal(S.from,           cardinals['South']),
               ... ) # omitted for brevity

@soid.register
def F(E, S):
    return ...       # omitted for brevity

@soid.register
def behavior(D):
    return Equal(D.move, False)
```

Fig. 3. A counterfactual specified using soidlib for a simplified grid-based car crash implementation (also available within our codebase alongside our soid-gui). This query leaves the turn signal of the 'other' car at $(2, 1)$ unconstrained, defining a counterfactual family. The objects `E`, `S`, and `D` are user-specified in an omitted `declare` function, including datatype.

Before working with soid, the investigator must use their domain expertise to find and extract the critical moment they care about from the factual trace within the logging infrastructure of A. We assume some mechanism guarantees the authenticity of the trace, such as an accountable logging protocol, as has been previously proposed for cyberphysical systems [33]. After extracting the trace the investigator must specify the i) (counter)factual query defining the factual, counterfactual, or family of counterfactual scenarios the query concerns; as well as ii) some possible agent behavior. In the remainder of this section, we explain how the user does so using soid and a Python library interface it exposes called soidlib. Constraints are specified through an API similar to Z3Py, see Fig. 3, while queries can be written as independent Python scripts or generated dynamically within a Python codebase.

Upon invocation, soid symbolically executes A to generate a set of feasible program paths as constrained by the (counter)factual query. The constraints in that query must be provided directly to the symbolic execution engine – an integration API exposes the query to the symbolic execution in order to enable this communication, or the user can do so directly outside soid itself. After the symbolic execution completes, soid formulates the query formula and invokes Z3 to resolve it. It then outputs to the user the finding, as well as any model – which exists in the event of a failed 'would' or successful 'might' query.

Query API. The query API of soid is exposed as a Python library called soidlib. A query specified using soidlib is composed of a name and query type, as well as a set of functions. These functions return either soidlib variable declarations or constraints, which are in either case automatically encoded into a set of corresponding Z3Py constraints for use during SMT solving to establish the satisfiability or validity of the query. An example query is shown in Fig. 3. The main API function interfaces the user must define in order to encode their query are:

- `declare()`: A function that must return three dictionaries of soidlib variable declarations, enumerating the set of environmental inputs (`E`) and internal state inputs (`S`) over which the factual or (family of) counterfactual scenario(s) are defined, as well as the set of decision (`D`) variables over which the behavior is defined. In order to do this soidlib exposes a variety of variable types, which it then converts into Z3 statements with the appropriate logical sorts as required by the underlying SMT logic (*e.g.*, encoding an object of integer type as an object of the 32-bit bitvector sort).
- `environmental(E)`: A function that must return a soidlib constraint over E describing the environmental program inputs.
- `state(S)`: A function that must return a soidlib constraint over S describing the internal state program inputs.
- `falsified(E, S)`: An optional function, returns a soidlib constraint encoding a concrete factual to be negated from the query formula, and therefore excluded from the set of possible output models.
- `behavior(D)`: A function that must return a soidlib constraint over D describing the behavior being queried.

Language Support. Through a modular API soid extensively supports any symbolic execution engine that produces output in the SMT-LIB format [4]. An integrator needs only to write a Python class implementing an interface between soid and the engine. As such, soid supports agents written in any programming language for which a suitable symbolic execution engine is available. We use the KLEE family of symbolic execution engines throughout our benchmarks. At present, support is integrated into soid for C language programs with floating-point instructions using KLEE-Float [21], working over the SMT logic of `QF_FPBV`, the quantifier-free theory of floating-point and bitvectors. Support is also integrated for C and C++ language programs without floating-point using mainline KLEE [5], producing representations in `QF_ABV`, the quantifier-free theory of arrays and bitvectors.[1] KLEE can be further extended to analyze other LLVM-compilable languages such as Rust [22], while other engines exist for compiled binaries [29] and many other languages including Java [2] and Javascript [23].

Symbolic Execution API. Adding support for a new symbolic execution engine to soid requires specifying between two and five functions: `preprocess`, `execute`, `parse`, `clean`, and `postprocess`, which are all hooked into the main soid execution path. Only `execute` and `parse` are necessary – they must respectively invoke the symbolic execution and then process the output into a list of Z3Py statements capturing the possible path conditions. Optionally, `clean` provides a hook for cleaning up temporary or output files generated by the symbolic execution engine, while `preprocess` and `postprocess` are designed

[1] Adding support for floating-point instructions into mainline KLEE remains at present an open enhancement for the project, see: https://klee.github.io/projects/.

to automate additional steps that may be desirable for the symbolic execution – the former is given access to the query, the latter additionally to the set of variables declared along the path conditions. For example, KLEE-Float automatically converts arrays into bitvectors using a technique called Ackermannization [25], and renames any such variables in the process. The KLEE-Float `preprocess` function packaged with soid i) casts objects as necessary; and ii) constrains them to equal the corresponding input declarations in the `declare` function so that they alias those inputs, *e.g.*, adding the constraint `(= (fp.to_ieee_bv data) data_ackermann!0)` where `data_ackermann!0` is KLEE-Float's synthesized, Ackermannized representation of `data`.

Query to Symbolic Execution. One of the major benefits of the *ex post* method of soid is that the (counter)factual query specified by the user can be used to constrain what parts of the program A are relevant to the scenarios in question and therefore must be included in the formula being checked. However, in order to do so the query must also be exposed to the symbolic execution engine in order to limit the symbolic execution to just the (ideally small) set of program paths feasible under the (counter)factual scenario conditions. This can either be done independent of soid, *e.g.* by the code invoking soid when it is used as a library, or by using the `preprocess` hook in the symbolic execution framework. At present, our codebase exclusively uses the external method.

Invocation. There are two ways to use soid: through a command line script (the `soidcli`) or directly as a Python library. If the latter, the user calling the code must declare a `soid.Oracle` object and configure it with i) a `soid.Query`; ii) the path to the A; and iii) the identity of the symbolic execution engine. If using the `soidcli`, the CLI script declares the oracle object for the user, who must specify the path to where (a collection of) `soid.Query` objects can be found declared in independent Python scripts (as well as the same path to A and symbolic execution engine identity). In case multiple variants of A are required in order to specify different symbolic execution preconditions for different counterfactual families, soid passes an identifier corresponding to a `priority` index that the user can specify through the CLI interface. In the examples present in the soid codebase this identifier is passed to a Makefile, which is then used to invoke KLEE(-Float) on the correct variant.

2.1 Example #1: Decision Tree Inference

Using soid, we analyzed our decision tree misclassification motivating example. The results are summarized in Table 1, and were gathered on an Intel Xeon CPU E5-2650 v3 @ 2.30GHz workstation with 64 GB of RAM. We used scikit-learn [27] to train a decision tree over the Pima Indians dataset as used in [31]. We then implemented A as a C program that preprocesses the data – triggering the software system bug, as it does so without the necessary unit conversion – and then infers a binary classification using the decision tree. In order to create

Table 1. Benchmark results for our incorrect statistical inference example.

		timings (avg. $n = 10$)			
model	**output**	**symbolic** (s)	**solving** (s)	**total** (s)	**paths**
$\rightarrow$	φ_{fact}, *low risk?*				
dt	✔	0.746	4.896e-03	0.812	1
$\square\!\!\rightarrow$	$\varphi^* \equiv \varphi_{fact}[(\texttt{weight} = 249.973) \mapsto \top]$, *ever high risk?*				
dt	✔	2.277	1.655	4.009	2

the factual basis for an investigation, we then invoked A on an example input where the unit conversion bug leads to the misclassification of the input as *low risk* instead of *high risk*.

We posed two queries:

1. Did the classification happen as described?
2. Does there exist a *weight* input parameter for which the instance is instead classified as *high risk* instead?

The former query provides a baseline for how much the counterfactual possibility of the latter query increases the cost of solving. It also fulfills the natural goal of many accountability processes to formally confirm apparent events and create a confirmed, end-to-end chain of analysis so that there is the highest possible societal confidence in any policy changes or punishments derived it. Both of these queries were resolved by soid in the positive, requiring at most a few seconds, even over a program structure in A that includes recursive invocations of floating-point comparison operations. Together, they demonstrate the weight input to A was causal for the classification, and establish its lack of unit conversion as contributory to the (harmful) misclassification decision.

Working with A, soid provides an adaptive oracle allowing the investigator to query its behavior and receive prompt and useful answers. The output of the program is also simple and interpretable. Without an intermediating GUI or developer tools, soid does require comfort with its API and the logical framework of expressing (counter)factuals and program outputs, but we do not expect a usable interface would be meaningfully difficult to integrate for this example.

3 soid-gui Architecture and Usage

The soid-gui is a web-based interactive interface for soid applied to the domain of autonomous vehicle accountability. It demonstrates that the use of soid can be managed by a high-level abstraction that exposes to non-technical practitioners the expressiveness and capacity of the tool, but none of its logical or technical complexity. We demonstrate the design and use of the soid-gui in Fig. 4.[2]

[2] The repositories for soid and soid-gui are available at https://github.com/sjudson/soid and https://github.com/mattelacqua/duckietown-soid, respectively.

Architecturally, the soid-gui is composed of three main components: i) a frontend written in React; ii) a backend server written in Python that operates a vehicle simulation using the Duckietown simulator for the OpenAI Gym (henceforce Gym-Duckietown [6]) and also interfaces with soid; and iii) a proxy server that manages communication between the browser frontend and the server backend. The Duckietown simulation is used as a stand-in for the real vehicle logs and instrumentation on which soid would be deployed in practice. We designed the crossroads intersection simulation interface to mimic the real-time driving context interface generated by contemporary autonomous vehicles, like those produced by Tesla. We stress that Gym-Duckietown is not exposed to soid, which operates exclusively over the program (and decision model) A. Gym-Duckietown is used only to simulate crashes and generate logfiles as the basis for soid queries.

Outside of the soid investigatory loop, the user can first use the soid-gui to design a car crash scenario by manipulating the location, destination, and other properties of the simulated car through menus and a drag and drop interface (see Fig. 4). The soid-gui also allows the user to select from among five different decision logics for the ego car: a directly programmed 'ideal' car, and four reinforcement-learned (specifically, Q-learned [32]) agents, colloquially the 'defensive', 'standard', 'reckless' and 'pathological' decision models. They are so named on the basis of the reward profiles used to train them.

After an iteration of the simulation (usually, after a crash occurs), the soid-gui allows the user to operate the soid investigatory loop. Using a slider the user can pick out a moment from the logs of the agent, and supported by detailed logging information about the inputs to A at each timestep can select the critical moment (see Step 1 in Fig. 4). They can then use car-specific dropdown menus to specify counterfactuals about any of the agents in the system in a user-friendly manner, which fully abstracts away the underlying logical formalism (Step 2 in Fig. 4). Finally, they can invoke soid on the query they have specified by asking whether the ego car 'might' or 'would' move or stop under the (family) of counterfactual scenario(s) they have defined (Step 3 in Fig. 4). After solving the soid-gui then presents an interpretable answer, including a valuation for any variables the counterfactual was stated over when one is available (Step 4 in Fig. 4). The user can then clear or adjust their counterfactual statement and ask further queries, until satisfied they have reached an understanding of the car's decision making under the selected decision model.

To use soid, the soid-gui first writes out a C language file with the necessary constraints for the KLEE-Float symbolic execution. It then creates the `soid.Query` and `soid.Oracle` objects, allowing it to invoke soid through the Python library interface. Once soid has invoked KLEE-Float and Z3 to determine the answer to the query the output is then processed. When applicable, this includes model parsing. The result is then passed back to the browser frontend to be shown to the user.

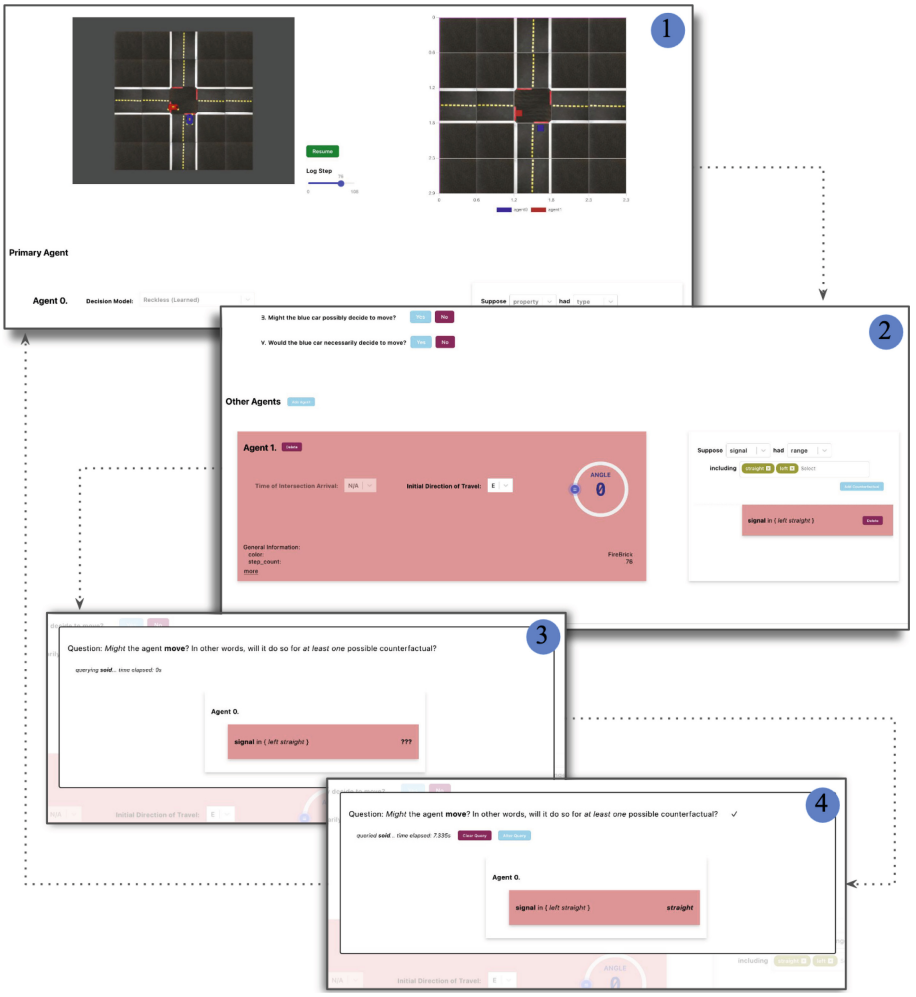


Fig. 4. After the (simulated) execution, the investigator (1) selects a critical moment; (2) poses a counterfactual query; (3) invokes the SMT solver; and (4) is presented with the response from the oracle.

3.1 Example #2: Three Cars on the Stand

We use the soid-gui to investigate a crash in Fig. 4. It is a simple intersection scenario, where the blue 'ego' car under investigation strikes the broadside of the red 'other' car which has indicated a right turn but proceeded straight nonetheless. As the red car possesses the right of way the fault lies with the blue car. We investigate 'to what purpose' the blue car entered in the intersection, in order to grade the severity of its misconduct in conjunction with legal norms that frequently apply the greatest possible penalties to purposeful action [16]. Notably,

this crash occurs for all three of the 'standard', 'reckless', and 'pathological' decision models (but *not* the 'defensive' model).

Table 2. Benchmark results for our car crash example. For the final query, we phrased it as both a 'would' and a 'might' counterfactual for comparison.

		timings (avg. $n = 10$)			
model	**output**	**symbolic** (s)	**solving** (s)	**total** (s)	**paths**
$\rightarrow$	φ_{fact}, *moved?*				
standard	✔	3.575	4.290e-03	4.162	1
impatient	✔	3.607	4.317e-03	4.193	1
pathological	✔	3.626	4.249e-03	4.212	1
$\square\!\!\rightarrow$	$\varphi^* \equiv \varphi_{fact}[(\texttt{agent1\_signal\_choice} = 2) \mapsto (\texttt{agent1\_signal\_choice} \in \{0, 1, 2\})]$, *always move?*				
standard	✘	3.979	2.371	7.754	3
impatient	✔	4.001	2.307	7.703	3
pathological	✔	3.958	2.326	7.681	3
$\square\!\!\rightarrow$	$\varphi^*[(\texttt{agent1\_pos\_x} = 1.376) \mapsto (1.0 \leq \texttt{agent1\_pos\_x} \leq 1.5)]$, *always move?*				
standard	✘	154.7	17.14	179.7	19
impatient	✔	207.6	4.622	220.1	19
pathological	✘	141.1	17.34	166.1	19
$\square\!\!\rightarrow$	$\varphi^* \wedge (\texttt{agent2\_pos\_x} = 1.316) \wedge (\texttt{agent2\_pos\_z} = 0.378) \wedge \cdots$, *always move?*				
standard	✘	8.995	4.111	16.74	3
impatient	✔	9.107	3.951	16.71	3
pathological	✔	9.037	3.913	16.54	3
$\square\!\!\rightarrow$	$\varphi^* \wedge (\texttt{agent2\_pos\_x} = 1.316) \wedge (\texttt{agent2\_pos\_z} = 0.378) \wedge \cdots$, *ever not move?*				
standard	✔	8.483	4.029	16.33	3
impatient	✘	8.979	3.848	16.46	3
pathological	✘	9.087	3.941	16.70	3

We pose three queries about the blue car's decision making at the moment when it releases the brakes and enters the intersection (Step 1 in Fig. 4):

1. Did the blue car actually decide to move, as it appeared to?
2. Could a different turn signal have led the blue car to remain stationary?
3. If the blue car had arrived before the red car and the red car was not signaling a turn, might the blue car have waited to 'bait' the red car into entering the intersection and creating the opportunity for a crash?

Intuitively, the second question should distinguish the 'standard' car from the 'reckless' and 'pathological', which should continue to move into the intersection no matter what. The third question should then distinguish between the 'reckless' and 'pathological' cars, with the former taking the opportunity for a clean path through the intersection, while the latter lies in wait.

There are natural explanations for the behavior of the other decision models: the 'standard' car is undertaking common human driving behavior given the

perception of an unobstructed path through the intersection, the 'reckless' car demonstrates a prioritization of individual speed over collective safe driving, while the 'pathological' car might be attempting to trigger a crash for insurance fraud. Notably, in the case of the 'reckless' car, we do not want to inherently describe that behavior as incorrect as verification methods might, such as any implementing [28]. It could be that exigent circumstances necessitate reckless behavior, and that the blue car not entering the intersection as fast as possible would trigger a greater harm than a minor crash.

The results of our benchmarks are summarized in Table 2. As before, all of the statistics were gathered on an Intel Xeon CPU E5-2650 v3 @ 2.30GHz workstation with 64 GB of RAM. Each heading in Table 2 describes a family of (counter)factual scenarios and behavior, as well as whether the query is a verification ('would...?') or counterfactual generation ('might...?') one. The rows list the decision model invoked within A, the answer as determined by the verification oracle, timings, and the total number of feasible paths.

We find that soid provides an interpretable and adaptive oracle allowing the investigator to query a sequence of counterfactuals without directly interacting with A or the machine learned-model underlying it. Most of our queries resolved within $< 20s$, providing effective usability. The results of the queries demonstrate the distinctive behaviors expected of the three conflicting purposes, allowing a capable investigator to distinguish them as desired.

4 Conclusion

We briefly conclude by considering some future directions for extensions to soid.

Supporting DNNs. Many modern machine-learned agents rely on models built out of deep neural network (DNN) architectures. Extending soid to support such agents – most likely by relying on recent innovations in symbolic execution for neural networks [12] and SMT-based neural network verifiers [17,18] – is a possible direction for increasing the utility of soid.

Programming Counterfactuals. Although soid is adaptive, that does not necessarily mean it needs to be interactive. A further possible direction would be to design a counterfactual calculus as the basis for a programming language that would invoke soid as part of its semantics. Such a language could potentially be the basis for formalizing legal regimes for which counterfactual analysis forms a critical component. A related direction would be to integrate with a scenario specification language like SCENIC from the VerifAI project [9,11] to add another layer of capability onto the specification of families of counterfactuals.

Acknowledgements. The authors thank Gideon Yaffe, Man-Ki Yoon, Cristian Cadar, and Daniel Liew. This work was supported by the Office of Naval Research (ONR) of the United States Department of Defense through a National Defense Science and Engineering Graduate (NDSEG) Fellowship, by the State Government of Styria, Austria - Department Zukunftsfonds Steiermark, by EPSRC grant no EP/R014604/1, and by NSF awards CCF-2131476, CCF-2106845, CCF-2219995, CCF-2318974, and CNS-2245344. The authors would also like to thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the programme Verified Software where work on this paper was undertaken.

References

1. Adadi, A., Berrada, M.: Peeking inside the black-box: a survey on Explainable Artificial Intelligence (XAI). IEEE Access **6**, 52138–52160 (2018)
2. Anand, S., Păsăreanu, C.S., Visser, W.: JPF–SE: a symbolic execution extension to Java PathFinder. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 134–138. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_12
3. Angwin, J., Larson, J., Mattu, S., Kirchner, L.: Machine Bias. ProPublica (May 23rd, 2016). https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version **2**, 6 (2021)
5. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI '08), pp. 209–224 (2008)
6. Chevalier-Boisvert, M., Golemo, F., Cao, Y., Mehta, B., Paull, L.: Duckietown Environments for OpenAI Gym. https://github.com/duckietown/gym-duckietown (2018)
7. Chockler, H., Halpern, J.Y.: Responsibility and blame: a structural-model approach. J. Artif. Intell. Res. **22**, 93–115 (2004)
8. Dastin, J.: Amazon scraps secret AI recruiting tool that showed bias against women. Reuters (2018). https://www.reuters.com/article/us-amazon-com-jobs-automation-insight/amazon-scraps-secret-ai-recruiting-tool-that-showed-bias-against-women-idUSKCN1MK08G
9. Dreossi, T., et al.: VerifAI: a toolkit for the formal design and analysis of artificial intelligence-based systems. In: Intentional Conference on Computer Aided Verification (CAV '19), pp. 432–442. Springer (2019)
10. Feigenbaum, J., Jaggard, A.D., Wright, R.N.: Accountability in computing: Concepts and Mechanisms. Found. Trends® Privacy Security **2**(4), 247–399 (2020)
11. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), pp. 63–78 (2019)
12. Gopinath, D., Wang, K., Zhang, M., Pasareanu, C.S., Khurshid, S.: Symbolic Execution for Deep Neural Networks. arXiv preprint arXiv:1807.10439 (2018)
13. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., Pedreschi, D.: A survey of methods for explaining black box models. ACM Comput. Surv. (CSUR) **51**(5), 1–42 (2018)
14. Halpern, J.Y., Pearl, J.: Causes and explanations: a structural-model approach. part i: causes. British J. Philos. Sci. **56**(4), 843–887 (2005)

15. Halpern, J.Y., Pearl, J.: Causes and explanations: a structural-model approach. part II: explanations. British J. Philos. Sci. **56**(4), 889–911 (2005)
16. Judson, S., Elacqua, M., Córdoba, F.C., Antonopoulos, T., Könighofer, B., Shapiro, S.J., Piskac, R.: 'Put the Car on the Stand': SMT-based Oracles for Investigating Decisions. In: ACM Symposium on Computer Science and Law (CSLAW '24) (2024). https://arxiv.org/abs/2305.05731 for an extended technical report
17. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: International Conference on Computer Aided Verification (CAV '17), pp. 97–117 (2017)
18. Katz, G.: The marabou framework for verification and analysis of deep neural networks. In: International Conference on Computer Aided Verification (CAV '19), pp. 443–452 (2019)
19. Kroll, J.A., et al.: Accountable algorithms. Univ. Pa. Law Rev. **165**(3), 633–705 (2017)
20. Lewis, D.: Counterfactuals. John Wiley & Sons (2013). originally published in 1973
21. Liew, D., Schemmel, D., Cadar, C., Donaldson, A.F., Zahl, R., Wehrle, K.: Floating-point symbolic execution: a case study in N-version programming. In: IEEE/ACM International Conference on Automated Software Engineering (ASE '17), pp. 601–612 (2017)
22. Lindner, M., Aparicius, J., Lindgren, P.: No panic! verification of rust programs by symbolic execution. In: 2018 IEEE 16th International Conference on Industrial Informatics (INDIN), pp. 108–114. IEEE (2018)
23. Loring, B., Mitchell, D., Kinder, J.: ExpoSE: practical symbolic execution of standalone JavaScript. In: International SPIN Symposium on Model Checking of Software (SPIN '17), pp. 196–199 (2017)
24. Mothilal, R.K., Sharma, A., Tan, C.: Explaining machine learning classifiers through diverse counterfactual explanations. In: ACM Conference on Fairness, Accountability, and Transparency (FAT* '20), pp. 607–617 (2020)
25. de Moura, L., Bjørner, N.: Model-based theory combination. Electron. Notes Theor. Comput. Sci. **198**(2), 37–49 (2008)
26. Moura, L.d., Bjørner, N.: Z3: An efficient SMT Solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08), pp. 337–340 (2008)
27. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
28. Shalev-Shwartz, S., Shammah, S., Shashua, A.: On a Formal Model of Safe and Scalable Self-Driving Cars. arXiv preprint arXiv:1708.06374 (2017)
29. Shoshitaishvili, Y., et al.: SoK: (State of) The art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy (S&P '16) (2016)
30. Smiley, L.: 'I'm the Operator': The Aftermath of a Self-Driving Tragedy. Wired Magazine (2022). https://www.wired.com/story/uber-self-driving-car-fatal-crash/
31. Wachter, S., Mittelstadt, B., Russell, C.: Counterfactual explanations without opening the black box: automated decisions and the GDPR. Harvard J. Law Technolo. **31**, 841 (2017)
32. Watkins, C.J.C.H., Dayan, P.: Q-learning. Mach. Learn. **8**, 279–292 (1992)
33. Yoon, M.K., Shao, Z.: ADLP: accountable data logging protocol for publish-subscribe communication systems. In: International Conference on Distributed Computing Systems (ICDCS '19), pp. 1149–1160. IEEE (2019)