Towards Causal Analysis of Protocol Violations

Shakil M. Khan and Mikhail Soutchanski

Department of Computer Science, Ryerson University, Toronto, Canada
{shakilmkhan,mes}@scs.ryerson.ca

Abstract. When a protocol specified within a given system fails to ensure some desired properties, it is important to identify the actual causes of this failure. In this paper, we utilize a formal model of causal analysis in the situation calculus to show how one can specify the actual causes of such violations in non-deterministic protocols defined within dynamic systems. We show that our definition has some desirable properties.

1 Introduction

Reasoning about violations in protocols is essential for many applications where it is important to design protocols that adhere to certain desirable properties [4, 9]. In case of a property violation, it is important to identify the actual causes of this failure. Such information can be used by the protocol designer to construct better protocols, e.g. by ensuring that certain execution paths are excluded. In this paper, we propose to utilize a formal model of causal analysis [2] in the situation calculus (**SC**) [12] to detect and reason about protocol violations.

We show how one can define the potential causes of protocol violations through the computation of causal chains within the SC. We make two assumptions: (1) there is a logical theory (with a complete initial state) that models how the system responds to actions, and (2) there is a non-deterministic protocol specified in the SC-based ConGolog programming language [5]. We are looking for events in all possible executions of the protocol to explain an observed effect.

Adopting a first-order language like the SC for causality analysis allows us to be more expressive. Namely, we can formulate quantified properties, model systems with infinite domains, and we can find violations in generic protocols specified over these systems. The underlying domain of objects in these systems can be infinite, e.g., it can include integer and real numbers with their standard interpretations. Furthermore, our formalization enables us to detect unwanted inter-component interactions in protocols, not just faulty component actions. We prove that our definition is sound and complete relative to a class of protocols.

2 Background

The Situation Calculus (SC). We use a version of the SC [12], where a dynamic domain is modeled using a basic action theory (**BAT**) $\mathcal{D}$ consisting of action precondition axioms (**APA**), successor-state axioms (**SSA**), initial state

axioms, unique name axioms for actions (**UNA**), and domain-independent foundational axioms Σ. We also utilize the single-step regression operator ρ. Given a query "does ϕ hold in situation $do(\alpha, s)$?", ρ transforms it into an equivalent query "does ψ hold in s?", eliminating action α by compiling it into ψ. The expression $\rho[\phi, \alpha]$ denotes such an equivalent query obtained from the formula ϕ by replacing each fluent atom F in ϕ with the rhs of the SSA for F where the action variable a is instantiated with the ground action α, and then simplified using UNAs and constants. One can prove that given a BAT $\mathcal{D}$, a formula $\phi(s)$ uniform in s, and a ground action term α, we have that $\mathcal{D} \models \forall s.\ \phi(do(\alpha, s)) \leftrightarrow \rho[\phi(s), \alpha]$.
Example. We use the well-known dining philosophers problem [7] with three philosophers as our running example. The actions in this domain are $pickUp(p, f)$, $putDown(p, f)$, $eat(p)$, while the fluents are $hasFork(p, f, s)$, $thinking(p, s)$, and $eating(p, s)$. Most of these and the following axioms are self-explanatory; see [8] for details. We define a philosopher p is waiting in situation s as an abbreviation $waiting(p, s) \stackrel{\text{def}}{=} \neg(eating(p, s) \vee thinking(p, s))$. We use the relation $neighb$ to describe the seating arrangement and assume the fork F_{ij} is in between philosophers P_i, P_j. We sample a few axioms (all free variables are $\forall$-quantified at front):

$(a).\ Poss(pickUp(p, f), s) \leftrightarrow$
$\quad \neg hasFork(p, f, s) \wedge \exists p'.(neighb(p, f, p') \vee neighb(p', f, p)) \wedge \neg hasFork(p', f, s),$
$(b).\ Poss(eat(p), s) \leftrightarrow \exists f, f'.\ f \neq f' \wedge hasFork(p, f, s)$
$\qquad\qquad\qquad\qquad \wedge hasFork(p, f', s) \wedge \neg eating(p, s),$
$(c).\ hasFork(p, f, do(a, s)) \leftrightarrow a = pickUp(p, f)$
$\qquad\qquad\qquad\qquad \vee (hasFork(p, f, s) \wedge \neg a = putDown(p, f)),$
$(d).\ eating(p, do(a, s)) \leftrightarrow a = eat(p) \vee (eating(p, s) \wedge \neg\exists f.\ a = putDown(p, f)),$
$(e).\ \forall p.\ thinking(p, S_0) \leftrightarrow p = P_1 \vee p = P_2 \vee p = P_3,$
$(f).\ \forall p, f.\ \neg eating(p, S_0) \wedge \neg hasFork(p, f, S_0).$

3 Actual Achievement Causes

Given a trace (a log), *actual achievement causes* are the key events responsible for achieving some effect. Here, we briefly review [2]. An effect is an SC formula $\phi(s)$ that is uniform in s. Given an effect $\phi(s)$, the actual causes of ϕ are defined relative to a *causal setting*, i.e., a BAT $\mathcal{D}$ representing the domain dynamics, and a narrative σ, representing the ground situation, where the effect was observed:
Def 1. *A causal setting is a tuple $\langle \mathcal{D}, \sigma, \phi(s) \rangle$, where $\mathcal{D}$ is a BAT, σ is a situation term of the form $do([a_1, \cdots, a_n], S_0)$ with ground action functions $a_1, \cdots, a_n$ s.t. $\mathcal{D} \models executable(\sigma)$, and $\phi(s)$ is an SC formula uniform in s s.t. $\mathcal{D} \models \phi(\sigma)$.*

As the theory $\mathcal{D}$ does not change, we will often suppress $\mathcal{D}$. We require ϕ to hold by the end of the narrative σ. Following [2], we identify the potential causes of an effect ϕ with a set of pairs, each of which consists of a ground action term occurring in σ and the situation where this action was executed. The notion of the achievement condition suggests that if some action α mentioned in σ triggers the formula $\phi(s)$ to change its truth value from false to true relative to $\mathcal{D}$, and if

there are no actions in σ after α that change the value of $\phi(s)$ back to false, then α is the actual cause of achieving $\phi(s)$ in σ. Batusov and Soutchanski [2] showed that when used together with the single-step regression operator ρ, this notion of achievement condition not only identifies the single action that brings about the effect of interest, but also captures recursively the actions that build up to it, i.e., the root causes. Additionally, one must include the preconditions under which these actions are executable. The following inductive definition formalizes this intuition. Let $\Pi_{apa}(\alpha, \sigma)$ be the right-hand side of the APA for action α with the situation term replaced by situation σ.

Def 2. *A causal setting $\mathcal{C} = \langle \sigma, \phi(s) \rangle$ satisfies the achievement of ϕ via the situation term $do(\alpha^*, \sigma^*) \sqsubseteq \sigma$ iff there is an action α' and situation σ' s.t.:*

$$\mathcal{D} \models \neg\phi(\sigma') \wedge \forall s.\, do(\alpha', \sigma') \sqsubseteq s \sqsubseteq \sigma \rightarrow \phi(s),$$

and either $\alpha^* = \alpha'$ and $\sigma^* = \sigma'$, or $\sigma^* \sqsubseteq \sigma' \sqsubset \sigma$ and the causal setting $\langle \sigma', \rho[\phi(s), \alpha'] \wedge \Pi_{apa}(\alpha', \sigma') \rangle$ satisfies the achievement condition via the situation term $do(\alpha^*, \sigma^*)$. Whenever a causal setting $\mathcal{C}$ satisfies the achievement condition via situation $do(\alpha^*, \sigma^*)$, we say that the action α^* executed in situation σ^* is an achievement cause in the causal setting $\mathcal{C}$.

Since the process of discovering intermediary achievement causes using ρ cannot continue beyond S_0, it eventually terminates. Moreover, since the narrative σ is finite, the achievement causes of $\mathcal{C}$ also form a finite sequence of situation-action pairs, which we call the *achievement causal chain of $\mathcal{C}$*.

Example (cont'd). Let the philosophers P_1, P_2, P_3 sit around the table, with forks in between. Consider the trace $\sigma_1 = do([pickUp(P_1, F_{12}), pickUp(P_3, F_{23}), pickUp(P_1, F_{13}), eat(P_1)], S_0)$. We are interested in computing the actual causes of the effect $\phi_1 = eating(P_1, s)$. Then according to Def. 2, the causal setting $\langle \phi_1, \sigma_1 \rangle$ satisfies the achievement condition ϕ_1 via the situation $do(eat(P_1), S_3)$, where $S_3 = do([pickUp(P_1,F_{12}), pickUp(P_3,F_{23}), pickUp(P_1,F_{13})],\ S_0)$, so the action $eat(P_1)$ executed in S_3 is a (primary) achievement cause of ϕ_1.

Moreover, computing $\rho[eating(P_1, \sigma_1), eat(P_1)] \wedge Poss(eat(P_1), S_3)$ yields $\exists f, f'.hasFork(P_1,f,s) \wedge hasFork(P_1,f',s) \wedge f \neq f' \wedge \neg eating(P_1, s)$ (let us call this formula ψ), and leads to a new causal setting $\langle S_3, \psi \rangle$. This satisfies the achievement condition via the action $pickUp(P_1, F_{13})$, so $pickUp(P_1, F_{13})$ executed in $S_2 = do([pickUp(P_1,F_{12}), pickUp(P_3,F_{23})], S_0)$ is a secondary achievement cause. Similarly, it can be shown that $pickUp(P_1,F_{12})$ executed in S_0 is also included in the causal chain. Notice that the action $pickUp(P_3,F_{23})$ is irrelevant.

We can also handle quantified queries, e.g. the actual causes of $\langle \sigma_2, \forall p.\ waiting(p, s) \rangle$, where $\sigma_2 = do([pickUp(P_1,F_{12}), pickUp(P_2,F_{23}), pickUp(P_3,F_{13})], S_0)$. Note that the integer-valued weight of pasta in the bowl can be easily modelled.

4 Causal Analysis of Protocol Violations

We model the behaviour of the system to be reasoned about as a BAT $\mathcal{D}$, while we encode an *observation* or *effect* using an SC formula $\phi(s)$ that is uniform in s, as above. For reasons explained below, we require the initial theory $\mathcal{D}_{S_0}$ to be complete both for relational and functional fluents. Let the protocol be specified

using a ConGolog program δ [5], but we can work with any programming language defined on top of the SC. We are now ready to give our formal definition of the potential causes of a protocol violation in the SC:

Def 3. *Given a system $\mathcal{DS} = \langle \mathcal{D}, \delta, \phi(s) \rangle$, the causes of violation of $\mathcal{DS}$ is the least set of causal chains $\mathcal{V}_{\mathcal{DS}}$ such that if there is a ground sequence of actions $\boldsymbol{a}$ for which $\mathcal{D} \models Do(\delta, S_0, do(\boldsymbol{a}, S_0)) \wedge \phi(do(\boldsymbol{a}, S_0))$, then $\mathcal{V}_{\mathcal{DS}}$ includes the causal chain relative to the causal setting $\langle \mathcal{D}, do(\boldsymbol{a}, S_0), \phi(s) \rangle$.*

Thus, the causes $\mathcal{V}_{\mathcal{DS}}$ of violating a non-deterministic protocol δ specified within a dynamic system $\mathcal{D}$ relative to a property $\phi(s)$ is the set of causal chains over all possible undesirable executions of δ, i.e. terminated executions over which $\phi(s)$ holds. Subsequently, we also call $\mathcal{V}_{\mathcal{DS}}$ a set of *conjectures*. Each conjecture specifies what actions in what situations should have been avoided by the executer, i.e. which paths in the execution tree of δ should have been prohibited by the protocol in an attempt to avoid failure $\phi(s)$. If the number of possible terminated executions of δ is finite, then $\mathcal{V}_{\mathcal{DS}}$ is also finite.

Note that Def. 3 may produce unintuitive results if the initial theory is incompletely specified. To see this, consider the non-deterministic program $(A|B)$, where the preconditions of A is $F(s)$ and that of B is $\neg F(s)$, and both A and B executed in S_0 have the effect that $\phi(s)$. Suppose that $\mathcal{D}$ does not specify the truth value of F in S_0. Although both A and B are the causes for $\phi(s)$, the theory $\mathcal{D}$ entails neither $executable(do(A, S_0))$, nor $executable(do(B, S_0))$, and therefore, according to our definition, the set of conjectures is empty, which is unintuitive. To avoid this issue, we require $\mathcal{D}$ to be initially complete.

Example (cont'd). Consider a simple protocol δ_1 specified in ConGolog:

$$(pickUp(P_1, F_{12}) \mid pickUp(P_2, F_{12})); (pickUp(P_1, F_{13}) \mid pickUp(P_2, F_{23})); \\ (eat(P_1) \mid eat(P_2) \mid \pi f.\, [Poss(pickUp(P_3, f), now)?; pickUp(P_3, f)]).$$

That is, first, either philosopher P_1 or P_2 non-deterministically picks up the fork F_{12} that is between them, then either of them picks up another available fork, and finally either P_1 eats, or P_2 eats, or P_3 picks up a fork. We would like to check if δ_1 violates the property that $\phi_3(s) = \neg\exists p.\, eating(p, s)$.

It is easy to see that there are only six possible executions of δ_1 and only in two of these cases, a philosopher is eating. For instance, no philosopher is eating in $do(\boldsymbol{a_1}, S_0)$, where $\boldsymbol{a_1} = [pickUp(P_1, F_{12}), pickUp(P_2, F_{23}), pickUp(P_3, F_{13})]$. As such, $\mathcal{V}_{\mathcal{DS}}$ for our example includes the causal chain relative to setting $\langle \mathcal{D}, do(\boldsymbol{a_1}, S_0), \phi_3(s) \rangle$. Note that the information provided by the causes of violation can be used by the protocol designer to reason about and improve on the protocol, in this case e.g. by ensuring that the second *pickUp* action is only performed by the philosopher who is already holding another fork, etc.

Notice our approach can detect improperly synchronized inter-component interactions, as can be seen even in this simple example: while none of the philosophers failed to perform, in all four cases suggested by our causes of violation their actions are not synchronized relative to the fulfillment of $\neg\phi_3$.

We now show that our formalization has some intuitively desirable properties. First, a conjecture for a given complete execution of a protocol is unique:

Th 1. *If $\mathcal{K}_1$ and $\mathcal{K}_2$ are two conjectures of a particular execution $\boldsymbol{a}$ of protocol δ specified over a system $\mathcal{DS} = \langle \mathcal{D}, \delta, \phi(s) \rangle$, then $\mathcal{K}_1 = \mathcal{K}_2$.*

Moreover, the set of actions in a conjecture is sufficient for the effect to hold.
Th 2. *If $\mathcal{K}$ is a conjecture of an execution $\boldsymbol{a}$ of protocol δ specified over a system $\mathcal{DS} = \langle \mathcal{D}, \delta, \phi(s) \rangle$, and $\sigma_{\mathcal{K}}$ is the situation obtained by performing the actions in $\mathcal{K}$ in the order they appear in $\boldsymbol{a}$ starting from S_0, then $\mathcal{D} \models executable(\sigma_{\mathcal{K}}) \wedge \phi(\sigma_{\mathcal{K}})$.*
Th. 1, 2, and Def. 3 together imply that our notion of causes of protocol violation is sound in the sense that each conjecture represents one or more undesirable executions of δ and correctly identifies the underlying reasons for the effect.

However, perhaps somewhat surprisingly, we can show that not every action in a conjecture is necessary for the effect to follow. Let $\sigma_{\boldsymbol{a}}^{\overline{a',s'}}$ denote the situation that can be obtained by executing the exact sequences of actions as in $\boldsymbol{a}$ starting in S_0, except for action a' in situation s'.
Th 3. *There is a system $\mathcal{DS} = \langle \mathcal{D}, \delta, \phi(s) \rangle$, an action a', and a situation s', s.t. if $\mathcal{K}$ is a conjecture in $\mathcal{V}_{\mathcal{DS}}$ of a particular execution $\boldsymbol{a}$ of protocol δ and a' executed in s' is a cause in the conjecture $\mathcal{K}$, then: $\mathcal{D} \not\models \neg(executable(\sigma_{\boldsymbol{a}}^{\overline{a',s'}}) \wedge \phi(\sigma_{\boldsymbol{a}}^{\overline{a',s'}}))$.*
Thus, removing a cause a' in s' from the execution/trace $\boldsymbol{a}$ itself may not have any effect on $\phi(s)$ as it may be the case that another action on the trace restores the executability and/or brings about the effect, e.g. one that is currently being preempted by the cause. In fact this shows that our base framework does not choose an action as a cause when its effects are preempted by some earlier action.

Furthermore, we can show that the notion of modularity from [8] can be adapted to protocol violations, if one sub-divides the system into constituents.

Finally, we show that our notion of causes of protocol violation is complete with respect to a class of protocols, where each protocol δ_f has the following properties: each complete execution of δ_f is finite, and there is a finite number n of terminated executions of δ_f. The above assumptions can apply even if the underlying object domain is infinite, e.g., if in our example, there are fluents for the weight or the number of pasta in a bowl. If $\boldsymbol{a}$ is a sequence of actions, then let $\boldsymbol{a}_!$ denote any subsequence of this sequence that possibly omits some actions from $\boldsymbol{a}$ but does not alter the order of the actions in $\boldsymbol{a}$. Also, if $\mathcal{V}_{\mathcal{DS}}$ is the causes of violation of a system $\mathcal{DS}$, let $\mathcal{V}_{\mathcal{DS}}^{act}$ be the set that replaces each conjecture/causal chain in $\mathcal{V}_{\mathcal{DS}}$ with the sequential composition of the actions in the causal chain without changing the order of occurrence of these actions. We can prove the following:
Th 4 (Completeness). *If $\mathcal{V}_{\mathcal{DS}}$ is the causes of violation of a system $\mathcal{DS} = \langle \mathcal{D}, \delta_f, \phi(s) \rangle$, then there are no sequences of actions $\boldsymbol{a}$ and subsequence $\boldsymbol{a}_!$ such that $\mathcal{D} \models Do(\delta_f, S_0, do(\boldsymbol{a}, S_0)) \wedge \phi(do(\boldsymbol{a}, S_0))$ and $\boldsymbol{a}_! \notin \mathcal{V}_{\mathcal{DS}}^{act}$.*

5 Discussion

We emphasize that our formalization supports domains with infinitely many objects. This makes our work fundamentally different from approaches based on model checking [1]. Perhaps the closest work to ours that can be found in the

literature is by Datta et al. [4], who proposed a framework for determining accountability of security violations for tasks and protocols such as authentication and key exchange. Like us, they also use actual causes to determine accountability. A key difference between our work and theirs is that while their analysis is tied to the underlying application (simple programs, threads, etc.), our work is based on a formal model of causality in the SC; thus, it is more general.

In addition, there has been work on automatic verification of partial correctness of (Con)Golog programs, e.g., [10, 3], and [6]. These are mostly theoretical work. In contrast, our approach is more practical, since one can implement our causal analysis with the one-step regression operator using any off the shelf ConGolog interpreter that produces terminated executions.

Besides these, there has been practical work on checking partial correctness of Golog programs. For instance, [11] proposed mechanisms for automated verification of partial correctness of Golog programs using the notion of extended regression. The method has been implemented [11]. Examining how their approach would compare with our causal analysis-based approach is future work.

One limitation of our framework is that we assume that the initial state is completely specified. Also, currently we only deal with deterministic actions and discrete dynamic domains. Going beyond these limitations is future work.

Acknowledgement: This work was supported in part by the NSERC Canada.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
2. Batusov, V., Soutchanski, M.: Situation Calculus Semantics for Actual Causality. In: Proc. of AAAI Conf. on Artificial Intelligence, pp. 1744–1752 (2018)
3. Claßen, J., Lakemeyer, G.: On the Verification of Very Expressive Temporal Properties of Non-terminating Golog Programs. In: Proc. of ECAI, pp. 887–892 (2010)
4. Datta, A., Garg, D., Kaynar, D.K., Sharma, D., Sinha, A.: Program Actions as Actual Causes: A Building Block for Accountability. In: Proc. of IEEE Comp. Security Foundations Symp. (CSF), pp. 261–275 (2015)
5. De Giacomo, G., Lespérance, Y., Levesque, H.J.: ConGolog, A Concurrent Programming Language based on the Situation Calculus. Artificial Intelligence **121**(1-2), 109–169 (2000)
6. De Giacomo, G., Lespérance, Y., Patrizi, F., Sardiña, S.: Verifying ConGolog Programs on Bounded Situation Calculus Theories. In: Proc. of AAAI Conf. on Artificial Intelligence, pp. 950–956 (2016)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
8. Khan, S.M., Soutchanski, M.: Diagnosis as Computing Causal Chains from Event Traces. In: Proc. AAAI Fall Symp. on Integ. Plan., Diag., and Causal Res. (2018)
9. Leitner-Fischer, F., Leue, S.: Causality Checking for Complex System Models. In: VMCAI, Lecture Notes in Computer Science, v. 7737, pp. 248–267 (2013)
10. Liu, Y.: A Hoare-Style Proof System for Robot Programs. In: Proc. of AAAI/IAAI, pp. 74–79 (2002)
11. Mo, P., Li, N., Liu, Y.: Automatic Verification of Golog Programs via Predicate Abstraction. In: Proc. of ECAI, pp. 760–768 (2016)
12. Reiter, R.: Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press (2001)