Specification, Abduction, and Proof

Konstantine Arkoudas

MIT Computer Science and AI Lab
arkoudas@csail.mit.edu

Abstract. Researchers in formal methods have emphasized the need to make specification analysis as automatic as possible and to provide an array of tools in a uniform setting. Athena is a new interactive proof system that supports specification, structured natural deduction proofs, and trusted tactics. It places heavy emphasis on automation, seamlessly incorporating off-the-shelf state-of-the-art tools for model generation and automated theorem proving. We use a case study of railroad safety to illustrate several aspects of Athena. A formal specification of a railroad system is given in Athena's multi-sorted first-order logic. Automatic model generation is used abductively to develop from scratch a policy for controlling the movement of trains on the tracks. The safety of the policy is proved automatically. Finally, a structured high-level proof of the policy's correctness is presented in Athena's natural deduction calculus.

1 Introduction

Logic has been called "the Calculus of Computer Science" [9,20]: just as Calculus and differential equations can be used to model the behavior of continuous physical systems, the language of mathematical logic can be used as a succinct and unambiguous notation for specifying the structure and behavior of discrete systems. Once we have obtained a logical specification of such a system in the form of a logical formula $P_1 \wedge \cdots \wedge P_n$, we can begin to ask various questions:

1. Is the specification consistent? That is, does the formula $P_1 \wedge \cdots \wedge P_n$ have a model?
2. Does the specified system have a desired property P? That is, does the specification $P_1 \wedge \cdots \wedge P_n$ logically imply P?

Two different techniques are used to answer these questions: *model generation* and *theorem proving*. Model generation can be used to answer the first question positively by exhibiting a model for $P_1 \wedge \cdots \wedge P_n$. Theorem proving can be used to settle the second question positively by showing that the implication

$$P_1 \wedge \cdots \wedge P_n \Rightarrow P \tag{1}$$

is a tautology. On the flip side, we can use theorem proving to settle the first question negatively, by proving that the constant **false** follows logically from $P_1 \wedge \cdots \wedge P_n$; and we can use model generation to settle the second question negatively, by exhibiting a model for $P_1 \wedge \cdots \wedge P_n \wedge \neg P$, i.e., a *countermodel* for (1). In view of this natural synergy it would appear useful to have both of these techniques available in one uniform setting, and indeed several researchers have made such suggestions [22,10]. The system that we discuss in this paper, Athena, offers both.

Once we have concluded that our system has a desired property P, we are left with the task of explaining *why* that is—why P follows from $P_1 \wedge \cdots \wedge P_n$. One possible answer is: "Because our favorite theorem prover said so." That may be an acceptable answer, depending on the context of the application. But it provides no insight into our system and has little explanatory value. A much better alternative is to adduce a formal *proof* that shows how P is derived from the specification. Such a proof should be mechanically checkable in order to ensure its correctness. But it must also be *structured* [19]: It should be given in a natural deduction format, in a precisely defined language with a formal semantics, and should be expressed at a level of abstraction roughly equivalent to that of a rigorous proof in English. This brings us to our third topic, which is a major component of formal methods in its own right: the subject of *proof representation and checking.*

Athena [1] is a new system that integrates all three of these elements: model generation; automated theorem proving; and structured proof representation and checking. It also provides a higher-order functional programming language, and a proof abstraction mechanism for expressing arbitrarily complicated inference *methods* in a way that guarantees soundness, akin to the tactics and tacticals of LCF-style systems[1] such as HOL [8] and Isabelle [24]. Proof automation is achieved in two ways: first, through user-formulated proof methods; and second, through the seamless integration of state-of-the-art ATPs (such as Vampire [30] and Spass [31]) as primitive black boxes for general reasoning. For model generation, Athena integrates Paradox [6], a new highly efficient model finder. For proof representation and checking, Athena uses a block-structured Fitch-style natural deduction calculus [25] with novel syntactic constructs and a formal semantics based on the abstraction of *assumption bases* [2].

In this paper we will illustrate all these aspects of Athena with a case study. We will develop a policy for controlling the movement of trains in a railroad system and prove that the policy is sound, in the sense that it satisifies a certain notion of safety. The soundness of the policy is proved completely automatically by the off-the-shelf ATPs that Athena uses under the hood. ATP technology has made impressive strides over the last few years, and the systems that are bundled with Athena—especially Vampire, the winner of the last few CASC competitions—-are now remarkably efficient. To give some perspective, it took a senior CS student at MIT several solid hours of work to prove the main result of this case study; it took Vampire a fraction of a second.

[1] An important difference of such systems from Athena is that they are based on sequent calculi. By contrast, Athena uses a Fitch-style formulation of natural deduction [25], which helps to make proofs and proof algorithms more perspicuous.

Beyond the completely automatic verification of the policy's soundness, we also provide a structured proof for it in Athena's natural deduction framework, which is then successfully machine-checked (also in less than one second).

Moreover, we show that model generation is useful not only for consistency checking and for debugging our specifications, but also for building them. In particular, we demonstrate an aggressive use of model generation that performs *abduction* in a way that helps not only to debug a safety policy, but to build it in the first place.

In logical deduction, reasoning proceeds from the premises to the conclusion: We take a given number of propositions as premises and attempt to derive some desired conclusion from them. During system design, however, we are often faced with the problem in the reverse direction: We know the desired conclusion, but we are not sure what constraints would be required in order to ensure it. That is, we have a skeleton system description $P_1 \wedge \cdots \wedge P_n$ ready; and we have a desired property P. What we wish to know is what additional constraints $Q_1, \ldots, Q_m$ are necessary in order to guarantee P, i.e., such that

$$\{P_1, \ldots, P_n\} \cup \{Q_1, \ldots, Q_m\} \models P.$$

That is the problem of *abduction* [16,15], which proceeds in the reverse direction from deduction.

The following is a simple iterative procedure for this problem:

1. Set $C = \{\mathbf{true}\}$.
2. Try to prove $\{P_1, \ldots, P_n\} \cup C \models P$; if successful, halt and output C.
3. If unsuccessful, try to find a model for $\{P_1, \ldots, P_n\} \cup C \cup \{\neg P\}$.
4. If successful, use the information conveyed by that model to refine C appropriately and then loop back to step 2; if unsuccessful, fail.

We illustrate this algorithm in Section 3. The individual steps of the algorithm are semi-mechanical, as the corresponding problems are unsolvable; but, with the aid of highly efficient tools, steps 2 and 3 can be greatly automated. The fourth step is the one requiring the most creativity, but the *minimality* of the countermodels produced in step 3 is very useful here: on every iteration through the loop, the simplest possible countermodel is produced, and this greatly facilitates the conjecture of a general condition that rules out the countermodel. After a few iterations of successive refinement, we will eventually converge to an appropriate theory.

2 Specification of an Abstract Railroad Model

Our railroad model is based on an Alloy [13] case study by Daniel Jackson [12], which was in turn inspired by a presentation on modeling San Fransisco's BART railway by Emmanuel Letier and Axel val Lamsweerde at a meeting of IFIP Working Group 2.9 on Requirements Engineering in Switzerland, in February 2000.

We view a railroad abstractly by positing the existence of two domains `Train` and `Segment`. That is, we assume we have a collection of trains and a collection of track segments on the ground. Every segment has a beginning and an end, and motion on it proceeds in one direction only, from the beginning towards the end. Therefore, segments are unidirectional. Of course trains may move in opposite directions on different segments; but on any given segment trains move in one direction only. At the end of each segment there is a gate, which may be either open or closed. Gates will be used to control train motion.

2.1 Railroad Topology

Segments can be connected to one another, with the end of one segment attached to the beginning of another, and it is this connectivity that creates an organized railroad out of a collection of segments. We will capture this connectivity with a binary relation $\texttt{succ} \subseteq \texttt{Segment} \times \texttt{Segment}$. The intended meaning is simple: $\texttt{succ}(s_1, s_2)$ holds iff s_2 is a "successor" of s_1, i.e., iff the end of s_1 is connected to the beginning of s_2. A segment might have several successors. In general, multiple segments might end at the same junction and fork off into multiple successor segments. We stipulate that `succ` is irreflexive, so that no segment loops back into itself, and intransitive, which is an obvious physical constraint.

Two segments may *overlap*, meaning that there is some piece of track, however small, that is shared by both segments. Segments that cross, for instance, will be considered overlapping. We model this with a binary relation $\texttt{overlaps} \subseteq \texttt{Segment} \times \texttt{Segment}$. We will make two useful assumptions about this relation, reflexivity and symmetry. Clearly, both assumptions are consistent with the intended physical interpretation of `overlaps`. We thus have four axioms so far:

$$\forall\, s \,.\, \neg\texttt{succ}(s, s) \tag{2}$$

$$\forall\, s_1, s_2, s_3 \,.\, \texttt{succ}(s_1, s_2) \wedge \texttt{succ}(s_2, s_3) \Rightarrow \neg\texttt{succ}(s_1, s_3) \tag{3}$$

$$\forall\, s \,.\, \texttt{overlaps}(s, s) \tag{4}$$

$$\forall\, s_1, s_2 \,.\, \texttt{overlaps}(s_1, s_2) \Rightarrow \texttt{overlaps}(s_2, s_1) \tag{5}$$

2.2 Capturing the State of the System

How do we capture a configuration of the railroad system at a given point in time? In order to know the state of the system we need to know two things. First, the distribution of the trains on the segments. That is, for each train t we need to know what segment t is on. And second, for each segment, we need to know whether its gate is open or closed. For our purposes, the state will be completely determined by these two pieces of information. Accordingly, we posit a domain `State`, a function

$$\texttt{segOf} : \texttt{Train} \times \texttt{State} \rightarrow \texttt{Segment}$$

and a relation $\texttt{closed} \subseteq \texttt{Segment} \times \texttt{State}$. The interpretations are as stated above: $\texttt{segOf}(t, x)$ denotes the segment on which t is located in state x; and $\texttt{closed}(s, x)$ holds iff the gate of segment s is closed in state x.

It is useful to introduce an auxiliary relation $\texttt{occupied} \subseteq \texttt{Segment} \times \texttt{State}$ such that $\texttt{occupied}(s, x)$ holds iff segment s is "occupied" in state x. We define this explicitly as follows: $\forall\, s, x \,.\, \texttt{occupied}(s, x) \Leftrightarrow [\exists\, t \,.\, \texttt{segOf}(t, x) = s]$.

We will model train motion as a transition relation between states:

$$\texttt{reachable} \subseteq \texttt{State} \times \texttt{State}$$

The idea is that $\texttt{reachable}\,(x, y)$ ("state y is reachable from state x") iff y is identical to x except that some (possibly none) trains have moved to successor segments—provided of course that they could make such a move. Specifically:

$$\begin{aligned}&(\forall\, x, y)\ \texttt{reachable}\,(x, y) \Leftrightarrow\\ &\qquad [(\forall\, t)\ \texttt{segOf}(t, y) \neq \texttt{segOf}(t, x) \Rightarrow \texttt{succ}(\texttt{segOf}(t, x), \texttt{segOf}(t, y)) \wedge\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \neg\texttt{closed}(\texttt{segOf}(t, x))]\end{aligned}$$

That is, in going from state x to y, a train t either didn't move at all or else it had an open gate in state x and moved to a successor segment.

This relational formulation is highly non-deterministic and allows for any physically possible transition from one state to another,[2] including cases where only one train moves, where none do, where two or three of them do, etc. This non-determinism is desirable, since we want our model to cover as many scenarios as possible.

2.3 Safety

We will consider a state *safe* iff no two trains are on overlapping segments:

$$\forall\, x \,.\, \texttt{safe}(x) \Leftrightarrow [\forall\, t_1, t_2 \,.\, t_1 \neq t_2 \Rightarrow \neg\texttt{overlaps}(\texttt{segOf}(t_1, x), \texttt{segOf}(t_2, x))]$$

We can now ask what would be an appropriate policy for controlling train motion that would guarantee this safety criterion. We make this more precise as follows: We define a *sound safety policy* as any number of unary constraints on states $C_1, \ldots, C_n$ such that for all states x and y, *if*

1. x is safe;
2. x satisfies the constraints $C_1, \ldots, C_n$, i.e., $C_1(x), \ldots, C_n(x)$ hold; and
3. y is reachable from x

then y is also safe. The problem now is to come up with state constraints that constitute a sound safety policy in this sense.

[2] Modulo our simplifying assumptions, most notably, our assumption that moving from one segment to another is instantaneous.

3 Abduction via Model Generation

Initially we may well be at a loss in guessing what constraints might be appropriate. We will show how an efficient model finder can provide insight on how to proceed.

Let us start out with the most trivial state constraint possible: the constant **true**. With this policy, our safety statement becomes:

$$\forall\ x, y\ .\ [\mathtt{safe}(x) \wedge \mathtt{reachable}(x, y) \wedge \mathbf{true}] \Rightarrow \mathtt{safe}(y)$$

Athena uses a prefix s-expression syntax, so we can define this proposition as follows:

```
(define policy-safety
  (forall ?x ?y
    (if (and (safe ?x)
             (reachable ?x ?y)
             true)
        (safe ?y))))
```

Predictably, this statement isn't true, and when we try to prove it automatically by issuing the method call `(!prove policy-safety)`, we fail.

Now let us see why this does not hold. We will try to find a countermodel for this statement, and the details of that model will spell out why this trivial policy fails. Armed with that information, we can start developing a policy in increments by fixing the problems that are discovered by the model finder.

We start by issuing the following command:

```
(falsify policy-safety)
```
(6)

This command attempts to find a model for the collection of all the propositions in the current assumption base *plus* the negation of `policy-safety`. Within a few seconds, Athena informs us that a countermodel has been found, that is, a model in which all the propositions in the assumption base are true, but `policy-safety` is false. Athena displays the model by enumerating the elements of each sort and listing the extension of every function and predicate. In particular, command (6) results in the output shown in Figure 1.

The countermodel consists of two states, `state-1` and `state-2`. The second state is reachable from the first; and while the first state is safe, the second is not. Therefore, `policy-safety` is false in this model. The reason for the failure becomes evident when we inspect the details of the model. There are two segments, each of which is a successor of the other, and two trains. In `state-1`, `train-1` is on `segment-1` and `train-2` on `segment-2`, and *the gate of* `segment-1` *is open*. Consequently, `train-1` is free to move on to `segment-2`, and indeed in `state-2` we have both trains on the second segment—a violation of our safety notion. Graphically, the situation is depicted in Figure 2. We use small rectangular boxes to represent trains. An open (closed) gate is indicated by the symbol $\surd$ (respectively, $\times$).

The issue is this: when a successor of a segment s is occupied, then s ought to have a closed gate. This is clearly violated in the countermodel, and that is how the unsafe second state is obtained. Therefore, we formulate our first state constraint as follows:

$$C_1(x) \Leftrightarrow \forall\, s_1, s_2 \,.\, [\mathtt{succ}(s_1, s_2) \wedge \mathtt{occupied}(s_2, x) \Rightarrow \mathtt{closed}(s_1, x)]$$

for arbitrary x. Accordingly, we redefine `policy-safety` to be the following proposition: $\forall\, x, y \,.\, [\mathtt{safe}(x) \wedge \mathtt{reachable}(x, y) \wedge C_1(x)] \Rightarrow \mathtt{safe}(y)$.

When we try to prove this automatically, we fail, so we revert to the model finder. Issuing the command `(falsify policy-safety)` results in the countermodel shown in Figure 3 (we omit Athena's textual presentation of the model for space reasons). Once again, there are two states, where the first one is safe while the second one is reachable from the first but unsafe. There are three segments, s_1, s_2, and s_3, where $\mathtt{succ}(s_1, s_2)$, $\mathtt{succ}(s_2, s_3)$, and $\mathtt{succ}(s_3, s_1)$. Further, s_1 overlaps both s_2 and s_3, while s_2 and s_3 do not overlap. And there are two trains, t_1 and t_2. The first state is as depicted in the left half of Figure 3; namely, t_1 is on s_2, which has a closed gate, while t_2 is on s_3, whose gate is open. (Segment s_1 has a closed gate in this state, although that is immaterial since s_1 is not occupied in this state.) Note, in addition, that this state satisfies our constraint C_1. There is only one segment with an open gate, s_3. And C_1 allows s_3 to have its gate open because s_3 does not have any occupied successors. The only successor of s_3 is s_1, and there are no trains on s_1 in this state.

Now the unsafe second state, shown in the right half of Figure 3, is obtained from the first state when t_2 moves from s_3 to s_1. This is permissible because s_3 has an open gate in the first state. But the new state is unsafe because even though s_1 has only one train on it, it nevertheless overlaps with s_2, which is occupied by t_1. This violates our notion of safety, which prescribes a state safe iff there are no overlapping segments occupied by distinct trains. Since s_1 and s_2 are overlapping and occupied by distinct trains in the new state, the latter is unsafe.

```
succ(segment-1, segment-1) = false       overlaps(segment-1, segment-1) = true
succ(segment-1, segment-2) = true        overlaps(segment-1, segment-2) = false
succ(segment-2, segment-1) = true        overlaps(segment-2, segment-1) = false
succ(segment-2, segment-2) = false       overlaps(segment-2, segment-2) = true

                                         segOf(train-1, state-1) = segment-1
safe(state-1) = true                     segOf(train-1, state-2) = segment-2
safe(state-2) = false                    segOf(train-2, state-1) = segment-2
                                         segOf(train-2, state-2) = segment-2

closed(segment-1, state-1) = false       reachableFrom(state-1, state-1) = true
closed(segment-1, state-2) = true        reachableFrom(state-1, state-2) = true
closed(segment-2, state-1) = false       reachableFrom(state-2, state-1) = true
closed(segment-2, state-2) = false       reachableFrom(state-2, state-2) = true
```

Fig. 1. Athena output displaying a countermodel.

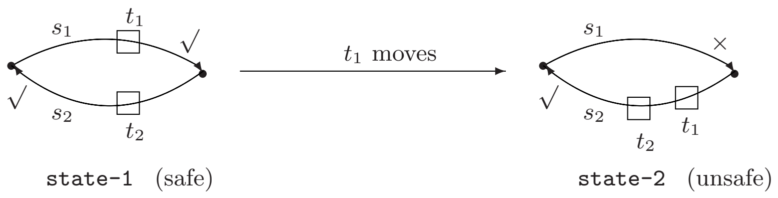

Fig. 2. A countermodel falsifying the trivial safety constraint *true.*

Thus we see that our initial constraint C_1 does not go far enough. It is not enough to stipulate that a predecessor of an occupied segment must have a closed gate; we must stipulate that a predecessor of a segment *that overlaps with an occupied segment* must have its gate closed. This is a stronger condition. It implies C_1, owing to our assumption that the overlaps relation is reflexive. So we introduce a new constraint C_1':

$$C_1'(x) \Leftrightarrow \forall\, s_1, s_2, s_3 \;.\; [\mathtt{succ}(s_1, s_2) \wedge \mathtt{overlaps}(s_2, s_3) \wedge \mathtt{occupied}(s_3, x) \Rightarrow \mathtt{closed}(s_1, x)]$$

and redefine `policy-safety` to be the proposition

$$\forall\, x, y \;.\; [\mathtt{safe}(x) \wedge \mathtt{reachable}(x, y) \wedge C_1'(x)] \Rightarrow \mathtt{safe}(y)$$

Unfortunately, when we attempt to prove this latest version automatically, we fail again. Returning to the model finder, we attempt to falsify this statement, which succeeds via the countermodel shown in Figure 4. As the picture makes clear, the problem is that two trains were able to move to the same segment simultaneously, because two distinct predecessors of the segment had open gates at the same time. To disallow this, we formulate the following constraint:

$$\begin{aligned}\forall\, x \;.\; C_2(x) \Leftrightarrow [\forall\, s_1, s_2 \;.\; s_1 \neq s_2 \wedge (\exists\, s \;.\; \mathtt{succ}(s_1, s) \wedge \mathtt{succ}(s_2, s)) \wedge \\ \neg\mathtt{closed}(s_1, x)] \Rightarrow \mathtt{closed}(s_2, x)\end{aligned}$$

This guarantees that, in any state, if two distinct segments have the same successor and one of them has an open gate, then the other will have a closed gate. This is an adaptation of the traffic rule which says that an intersection should

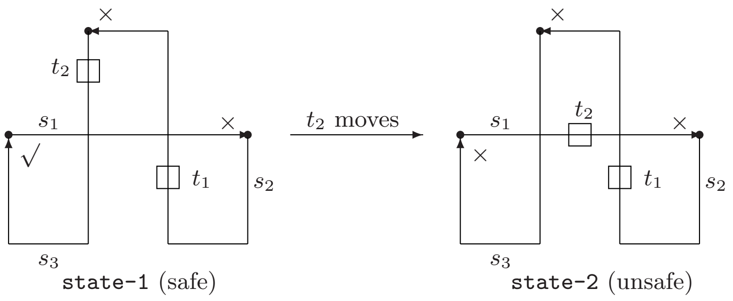

Fig. 3. A countermodel falsifying constraint C_1.

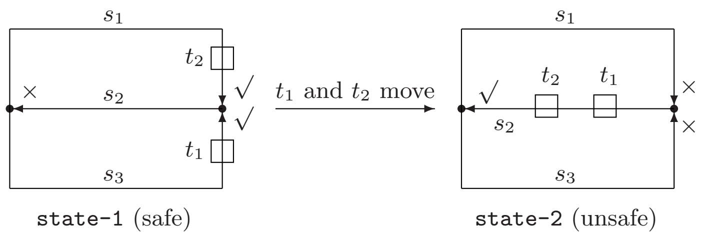

Fig. 4. A countermodel falsifying constraint C_1'.

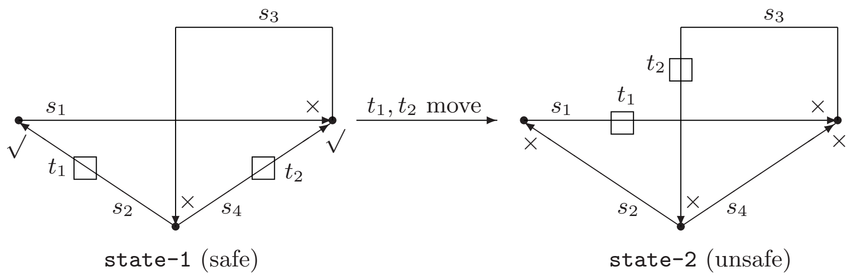

Fig. 5. A countermodel falsifying constraint C_2.

not show a green light in two different directions. We now redefine policy-safety as follows:

$$\forall\ x, y\ .\ [\texttt{safe}(x) \wedge \texttt{reachable}(x, y) \wedge C_1'(x) \wedge C_2(x)] \Rightarrow \texttt{safe}(y)$$

But this version is not valid either. Attempting to falsify it results in the countermodel shown in Figure 5. The problem is essentially a generalization of the situation depicted by the countermodel in Figure 4. This time, t_1 and t_2 do not move to the same segment, but to overlapping ones. This is possible because the segments on which t_1 and t_2 are placed initially (namely, s_2 and s_4) have overlapping successors and yet both of them have open gates at the same time. We need to prohibit this. Let us say that two distinct segments are *joinable* iff they have overlapping successors. That is, for all s_1 and s_2, $\texttt{joinable}(s_1, s_2)$ holds iff

$$s_1 \neq s_2 \wedge [\exists\ s_1', s_2'\ .\ \texttt{overlaps}(s_1', s_2') \wedge \texttt{succ}(s_1, s_1') \wedge \texttt{succ}(s_2, s_2')]$$

We then need to stipulate that of any two joinable segments, at most one has an open gate. We express this via a new constraint C_2' as follows:

$$C_2'(x) \Leftrightarrow [\texttt{joinable}(s_1, s_2) \wedge \neg\texttt{closed}(s_1, x) \Rightarrow \texttt{closed}(s_2, x)]$$

Observe that C_2' implies C_2 (since overlaps is reflexive), hence it is no longer necessary to state C_2. Therefore, our safety statement now becomes:

```
(define policy-safety
  (forall ?x ?y
    (if (and (safe ?x)
             (reachable ?x ?y)
             (C1' ?x)
             (C2' ?x))
        (safe ?y))))
```

This time, the attempt `(!prove policy-safety)` succeeds instantly, confirming that we have finally arrived at a sound safety policy.

4 Structured Proof Representation

We have automatically verified the soundness of our safety policy, and while that should boost our confidence in the policy, it is not quite good enough. As engineers, we should be able to convince others that our policy is indeed sound—we should be able to *justify* our policy with a solid argument. That justification should take the form of a rigorous mathematical proof. However, not just any proof will do. A formal proof of an important system property should serve as human-readable documentation: it should explain why the property holds. To that end, the proof should be structured, given in a natural deduction style resembling common mathematical reasoning, and at a high level of abstraction.

Athena proofs are expressed in a block-structured ("Fitch-style" [25]) natural deduction calculus. High-level reasoning idioms that are frequently encountered in mathematical practice are directly available to the user, have a simple semantics, and help to make proofs readable and writable. Athena's off-the-shelf ATP technology can be used to automatically dispense with tedious steps, focusing instead on the interesting parts of the argument and keeping the reasoning at a high level of detail.

Most interestingly, a block-structured natural deduction format is used not only for writing proofs, but also for writing tactics ("methods" in Athena terminology). This is a novel feature of Athena; all other tactic languages we are aware of are based on sequent calculi. Tactics in this style are considerably easier to write and remarkably useful in making proofs more modular and abstract. As this example will illustrate, writing methods can pay dividends even in simple proofs.

In what follows we present a formal Athena proof of the safety of our policy. As our starting point, and for purposes of comparison, consider first a rigorous proof of the result in English:

Theorem 1. *For all states x and y, if (1) x is safe; (2) y is reachable from x; and (3) x satisfies constraints C'_1 and C'_2; then y is also safe.*

Proof. Pick arbitrary states x and y and assume that x is safe; that y is reachable from x; and that $C'_1(x)$ and $C'_2(x)$ hold. Under these assumptions, we are to prove that y is safe.

We will proceed by contradiction. Suppose, in particular, that y is *not* safe. Then, by the definition of safety, there must be two distinct trains t_1 and t_2 on overlapping segments in y, that is, we must have $t_1 \neq t_2$ and

$$\texttt{overlaps}(\texttt{segOf}(t_1, y), \texttt{segOf}(t_2, y)) \tag{7}$$

We now ask: did either train move in the transition from state x to y, or did both stay on the same segment? Exactly one of these two possibilities must be the case, i.e., we must have either

$$\mathit{case}_1 \equiv [\texttt{segOf}(t_1, y) \neq \texttt{segOf}(t_1, x)] \vee [\texttt{segOf}(t_2, y) \neq \texttt{segOf}(t_2, x)]$$

(t_1 moved or t_2 moved); or else:

$$\mathit{case}_2 \equiv [\texttt{segOf}(t_1, y) = \texttt{segOf}(t_1, x)] \wedge [\texttt{segOf}(t_2, y) = \texttt{segOf}(t_2, x)]$$

(neither one moved). The disjunction $\mathit{case}_1 \vee \mathit{case}_2$ holds by the law of the excluded middle. We will now show that a contradiction ensues in either case.

Consider case_2 first, i.e., assume

$$\texttt{segOf}(t_1, y) = \texttt{segOf}(t_1, x) \tag{8}$$

$$\texttt{segOf}(t_2, y) = \texttt{segOf}(t_2, x) \tag{9}$$

Then, from (8), (9), and (7), we conclude

$$\texttt{overlaps}(\texttt{segOf}(t_1, x), \texttt{segOf}(t_2, x)) \tag{10}$$

i.e., that the segments of t_1 and t_2 *in state* x overlap. But t_1 and t_2 are distinct trains, so that would mean that state x is unsafe: that it has two distinct trains on overlapping segments. This is a contradiction, since we have explicitly assumed that x is safe.

Consider now case_1, where at least one of the trains has moved in the transition from x to y. Without loss of generality, assume that t_1 moved, so that

$$\texttt{segOf}(t_1, y) \neq \texttt{segOf}(t_1, x) \tag{11}$$

From this, along with the hypothesis that y is reachable from x and the definition of reachability, we conclude that the segment of t_1 in y is a successor of the segment of t_1 in x; and that the segment of t_1 in x had an open gate:

$$\texttt{succ}(\texttt{segOf}(t_1, x), \texttt{segOf}(t_1, y)) \tag{12}$$

$$\neg\texttt{closed}(\texttt{segOf}(t_1, x), x) \tag{13}$$

We now perform a case analysis depending on whether or not t_2 moved as well. Suppose first that, like t_1, t_2 also moved, so that:

$$\texttt{segOf}(t_2, y) \neq \texttt{segOf}(t_2, x) \tag{14}$$

As before, this entails (in tandem with the reachability of y from x) the following:

$$\texttt{succ}(\texttt{segOf}(t_2, x), \texttt{segOf}(t_2, y)) \tag{15}$$

$$\neg\texttt{closed}(\texttt{segOf}(t_2, x), x) \tag{16}$$

But this means that the segments $\texttt{segOf}(t_1, x)$ and $\texttt{segOf}(t_2, x)$ are joinable: (a) they are distinct (if they were identical, then x would be unsafe, since $t_1 \neq t_2$ and `overlaps` is reflexive, contrary to our assumption); and (b) they have overlapping successors (from (7), (12), and (15)). Therefore, state x has two joinable segments with simultaneously open gates—a condition that is explicitly prohibited by C_2', which is supposedly observed in x. Hence a contradiction.

By contrast, suppose that t_2 did not move during this state transition:

$$\texttt{segOf}(t_2, y) = \texttt{segOf}(t_2, x) \tag{17}$$

In that case, (7) entails

$$\texttt{overlaps}(\texttt{segOf}(t_1, y), \texttt{segOf}(t_2, x)) \tag{18}$$

This case violates constraint C_1' in state x: $\texttt{segOf}(t_1, x)$, the segment of t_1 in x, is the predecessor of a segment that overlaps with an occupied segment, namely, $\texttt{segOf}(t_2, x)$. Therefore, according to C_1', it should have its gate closed—but it does not, a contradiction.

This concludes the case analysis of whether t_2 moved, on the assumption that t_1 has moved. A symmetric argument can be given on the assumption that t_2 has moved. $\square$

This is a perfectly rigorous proof, with one exception: the phrase "without any loss of generality" is vague. Nevertheless, it is a frequent mathematical colloquialism. Typically, it means that there is a finite number of cases to consider $c_1, \ldots, c_n$, and it does not really make a difference which c_i we analyze because the reasoning for one of them can be readily applied to the others. This is reiterated in the closing remark that "a symmetric argument can be given on the assumption that t_2 moved."

These colloquialisms can be given more precise meaning with the help of algorithmic notions. What we really are saying above is that any proof for a particular c_i can be abstracted (over a number of appropriate parameters) into a general proof algorithm that can be just as well applied to the other cases. That is, we are claiming that there is a tactic that will produce the desired conclusion in any given case. In the Athena proof of the safety result, shown in Figure 6, we formulate such a method `M` that is capable of performing the correct analysis on a variable input assumption of which train has moved first. Treating both cases then becomes simply a matter of invoking `(!M t1 t2)` first and then transposing the arguments and invoking `(!M t2 t1)` for the second case.

The technical details of the Athena proof are not so important; the interested reader could follow them by consulting a description of the formal syntax and semantics of the language [4]. The important points are: (a) the Athena proof

```
(policy-safety BY
  (pick-any x y
    (assume-let ((hyp (and (safe x)
                           (reachable x y)
                           (C1A x)
                           (C2A x))))
      (!by-contradiction
         (assume (not (safe y))
           (dlet ((P (!derive (exists ?t1 ?t2
                                (and (not (= ?t1 ?t2))
                                     (overlaps (segOf ?t1 y) (segOf ?t2 y))))
                              [(not (safe y)) safe-definition])))
          (pick-witnesses (t1 t2) P
            (dlet ((t-property (and (not (= t1 t2))
                                    (overlaps (segOf t1 y) (segOf t2 y))))
                   (t-distinct (!derive (not (= t1 t2)) [t-property]))
                   (t-overlapping (!derive (overlaps (segOf t1 y) (segOf t2 y)) [t-property]))
                   (one-has-moved (!derive (or (not (= (segOf t1 y) (segOf t1 x)))
                                               (not (= (segOf t2 y) (segOf t2 x))))
                                           [hyp safe-definition t-property]))
                   (M (method (r1 r2)
                       (assume-let ((hyp1 (not (= (segOf r1 y) (segOf r1 x)))))
                         (dlet ((P1 (!derive (succ (segOf r1 x) (segOf r1 y))
                                             [hyp1 reachable-definition hyp]))
                                (P2 (!derive (not (closed (segOf r1 x) x))
                                              [hyp1 reachable-definition hyp]))
                                (c1 (assume-let ((case1 (not (= (segOf r2 y) (segOf r2 x)))))
                                      (dlet ((P3 (!derive (succ (segOf r2 x) (segOf r2 y))
                                                      [case1 reachable-definition hyp]))
                                             (P4 (!derive (not (closed (segOf r2 x) x))
                                                      [case1 reachable-definition hyp]))
                                             (P5 (!derive (not (= (segOf r1 x) (segOf r2 x)))
                                                       [hyp t-distinct safe-definition
                                                        (reflexive overlaps)]))
                                             (P6 (!derive (joinable (segOf r1 x) (segOf r2 x))
                                                      [P3 P1 t-overlapping P5
                                                       joinable-definition
                                                       (symmetric overlaps)])))
                                        (!derive false [C2'-definition P2 P4 P6 hyp]))))
                                (c2 (assume-let ((case2 (= (segOf r2 y) (segOf r2 x))))
                                      (dlet ((P7 (!derive (occupied (segOf r2 x) x)
                                                      [case2 occupied-definition]))
                                             (P8 (!derive (overlaps (segOf r1 y) (segOf r2 x))
                                                      [case2 t-overlapping
                                                       (symmetric overlaps)])))
                                        (!derive false [P7 P8 P2 hyp P1 C1'-definition
                                                       (symmetric overlaps)])))))
                           (!by-cases c1 c2 [])))))
                   (say-t1-moves ((if (not (= (segOf t1 y) (segOf t1 x))) false) BY (!M t1 t2)))
                   (say-t2-moves ((if (not (= (segOf t2 y) (segOf t2 x))) false) BY (!M t2 t1))))
              (!by-cases say-t1-moves say-t2-moves [one-has-moved])))))))))
```

Fig. 6. Athena proof of safety

reads more or less like the English proof, in that the overall *structure* of both proofs as well as the granularity of the individual inferences are similar; and (b) tactics in block-structured natural deduction style are easy to write and useful even in simple proofs.

5 Related Work

Alloy [13] is also aimed at automatic analysis of abstract system designs (mainly software systems). Alloy's specification language is based on Tarski's relational calculus, with heavy influences from Z [27]. It has been successfully used to analyze several systems, e.g. exposing bugs in Microsoft's COM [14] and in a naming architecture for dynamic networks [18]. The case study we presented in this paper was originally done in Alloy [12]. The main difference is that Alloy does not have a notion of proof, and hence it cannot be used to verify infinite-state systems. It can detect the presence of bugs but cannot establish their absence.

Prioni [3] is a tool that attempts to bridge this gap by integrating Alloy with Athena. Athena proofs in Prioni are based on an explicit first-order axiomatization of the calculus of relations and a lemma library containing various useful results for that calculus (e.g., that the transpose and reflexive transitive closure of a homogeneous binary relation commute). Essentially, the Alloy calculus of relations is used as an object language and Athena is used as a metalanguage to manipulate and reason about the object language. This indirection can complicate the reasoning required for the proof effort. Automation is also hindered in Prioni. For instance, the completely automatic soundness proof that we obtained for this case study would be highly unlikely in Prioni because the assumption base would be overpopulated with the entire axiomatization of the relational calculus and the lemma library. It is well known that ATPs get overwhelmed by large sets of premises [26]. Finally, Prioni's integration of Alloy and Athena is not seamless in the sense that the user must be fluent in both systems in order to use the tool. By contrast, in the approach we illustrated in this paper, the integration of Paradox into Athena is completely seamless; the user has no idea that Paradox is running under the hood.

This opaque use of Paradox is enabled by an automatic translation from Athena's multi-sorted logic to the standard single-sorted TPTP input format [28] of Paradox and back.[3] The translation is written in Athena itself, leveraging its facilities for manipulating propositions. Once a model has been produced, another translation takes place that transforms the Paradox output into a format that makes sense to the Athena user (as shown in Figure 1). Similar remarks can be made about Athena's use of ATPs. To take a simple example, Vampire has a fairly limited lexical notion of variables (e.g., they must begin with certain letters), whereas Athena is more liberal (e.g., names such as `?*d2-E3` are legal). Incompatibilities of this kind are resolved silently during the automatic translation. This approach pays heed to the lessons that have emerged from successful case studies, which stress that "evidence of the ATP must be hidden," and that due attention must be given to pragmatic issues, e.g., implementation restrictions such as reserved identifiers, length of symbols, etc.—issues that are "often overlooked in research-oriented environments" although "they are important prerequisites for successful applications of ATPs" [26].

Some ATPs such as Gandalf [29] and Spass [31] have model-building capabilities in addition to standard resolution-based refutation. However, direct application of such a batch-oriented ATP is challenging because of its "low bandwidth of interaction," which renders it "a toolkit with only one single tool" [7]. As Schumman puts it, such ATPs are like racing cars—they are fast and powerful but cannot be used in everyday traffic because essentials such as headlights are missing [26]. The same can be said about Paradox. By contrast, interactive proof environments (such as Isabelle, PVS, Athena, etc.) provide the essentials as well as bells and whistles: rich specification languages, definitional facilities for

[3] The standard embedding of many-sorted logic into single-sorted logic [21] is used in going from Athena to Paradox; this is safe for our purposes, since the embedding is well-known to preserve finite models.

incremental extension of proof scripts, abstraction mechanisms, computational capabilities, tactics, etc. It therefore makes more sense to harness the power of ATPs from within such environments rather than as stand-alone applications.

Theorem proving has been combined with model checking in systems such as PVS [23], ACL2 [17], SLAM [5], and others (SLAM's "counterexample-driven refinement" is also somewhat similar to our notion of incremental abduction via countermodel generation). Model checking is different from model generation—it checks whether a formula holds in a given model, so there are two possible answers: "valid" or "invalid," with the latter accompanied by an offending trace. The correctness of the model-checking algorithm implementation is crucial for the credibility of "valid" answers, as most model checkers do not emit proofs that can be independently checked (unlike ATPs such as Vampire and Spass, which do emit proofs). There have also been attempts to integrate some higher-order proof systems with first-order ATPs, as in the integration of Gandalf and HOL [11]. These are not entirely happy marriages, however, since some higher-order goals cannot be desugared into first-order logic; indeed, this remains an active research field. That is not an issue for Athena, which is a first-order system.

References

1. K. Arkoudas. Athena. `http://www.cag.csail.mit.edu/~kostas/dpls/athena`.
2. K. Arkoudas. Denotational Proof Languages. PhD dissertation, MIT, 2000.
3. K. Arkoudas, S. Khurshid, D. Marinov, and M. Rinard. Integrating model checking and theorem proving for relational reasoning. In *Proceedings of the 7th International Seminar on Relational Methods in Computer Science (RelMiCS 7)*, Malente, Germany, May 2003.
4. T. Arvizo. A virtual machine for a type-ω denotational proof language. Masters thesis, MIT, June 2002.
5. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *SPIN, Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings*, volume 2057 of *Lecture Notes in Computer Science*. Springer, 2001.
6. K. Claessen and N. Sorensson. New techniques that improve Mace-style finite model building. In *Model Computation—principles, algorithms, applications*, Miami, Florida, USA, 1973.
7. D. Cyrluk, S. Rajan, N. Shankar, , and M.K. Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design (TPCD '94)*, volume 901 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, sep 1994. Springer-Verlag.
8. M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic.* Cambridge University Press, Cambridge, England, 1993.
9. Joseph Y. Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. On the unusual effectiveness of logic in computer science. *The Bulletin of Symbolic Logic*, 7(2):213–236, 2001.
10. Constance L. Heitmeyer. On the need for practical formal methods. In *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 18–26. Springer-Verlag, 1998.

11. J. Hurd. Integrating Gandalf and HOL. In *Theorem proving in higher-order logics*, pages 311–321, 1999.
12. D. Jackson. Railway Safety. `http://alloy.mit.edu/case-studies.html`, 2002.
13. Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.
14. Daniel Jackson and Kevin Sullivan. COM revisited: Tool-assisted modeling of an architectural framework. In *Proc. 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, San Diego, CA, 2000.
15. J.R. Josephson and S.G. Josephson, editors. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge University Press, 1994.
16. A. C. Kakas and M. Denecker. Abduction in logic programming. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond. Part I*, number 2407 in Lecture Notes in Artificial Intelligence, pages 402–436. Springer-Verlag, 2002.
17. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.
18. S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *15th IEEE ASE*, 2000.
19. L. Lamport. How to write a proof. Research Report 94, Systems Research Center, DEC, February 1993.
20. Z. Manna and R. Waldinger. *The logical basis for computer programming*. Addison Wesley, 1985.
21. M. Manzano. *Extensions of first-order logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1996.
22. W. McCune. A Davis-Putnam program and its application to finite first-order model search. Technical Report ANL/MCS-TM-194, ANL, 1994.
23. S. Owre, N. Shankar, and J. M. Rushby. The PVS specification language (draft). Research report, Computer Science Laboratory, SRI International, Menlo Park, California, February 1993.
24. L. Paulson. *Isabelle, A Generic Theorem Prover*. Lecture Notes in Computer Science. Springer-Verlag, 1994.
25. F. J. Pelletier. A Brief History of Natural Deduction. *History and Philosophy of Logic*, 20:1–31, 1999.
26. J. Schumann. Automated theorem proving in high-quality software design. In Steffen Hölldobler, editor, *Intellectics and Computational Logic*, volume 19 of *Applied Logic Series*. Kluwer, 2000.
27. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., 1992.
28. C. Suttner and G. Sutcliffe. Technical Report on the TPTP Problem Library. `http://www.cs.miami.edu/~tptp/TPTP/TR/TPTPTR.shtml`.
29. T. Tammet. Gandalf. `http://www.cs.chalmers.se/~tammet/gandalf/`.
30. A. Voronkov. The anatomy of Vampire: implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2), 1995.
31. C. Weidenbach. Combining superposition, sorts, and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2. North-Holland, 2001.