Specification, abduction, and proof

Konstantine Arkoudas

RPI
arkouk@rpi.edu

Abstract. Researchers in formal methods have recognized the need to make formal specification analysis as automatic as possible and to provide a suite of analysis tools in a single uniform setting. Athena is a new formal system that seamlessly integrates specification, structured natural deduction proofs, trusted tactics, and cutting-edge off-the-shelf tools for model generation and automated theorem proving. We use a case study of railroad safety to illustrate all these 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" [4, 11]: just as calculus and analysis can be used to model the behavior of continuous physical systems, the language of mathematical logic can be used as a succinct, precise, and unambiguous notation for specifying the structure and behavior of digital systems. Once we have obtained a logical specification of a digital 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). It would thus appear useful to have both of these techniques available in a uniform setting, and indeed several researchers have made such suggestions [12, 5]. The system that we demonstrate in this paper, Athena, offers both.

Once we have concluded that our specified system indeed has a desired property P, we are left with the task of explaining *why* that is; i.e., why P follows from $P_1 \wedge \cdots \wedge P_n$. One possible answer is "Because our favorite theorem prover said so." This may be an acceptable answer, depending on the context of the application. But it provides no insight into our system and is of 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* [10]: it should be given in a natural deduction format, in a formal language with a precise but simple semantics, and should be stated at a level of abstraction roughly equivalent to that of a rigorous proof in English (most importantly, the individual proof steps should not be overly tedious). We are thus led to our third topic, which is a major component of formal methods in its own right: the subject of *proof representation and checking*.

Athena is a new system that provides all three of these capabilities: 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 such as HOL [3] and Isabelle [13]. 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 [15] and Spass [16] as primitive black boxes for general reasoning. For model generation, Athena integrates Paradox [2], a new highly efficient model finder. For proof representation and checking, Athena uses a block-structured Fitch-style natural deduction calculus [14] with novel syntactic constructs and a formal semantics based on the abstraction of *assumption bases* [1].

In this paper we will illustrate all of these aspects of Athena with a case study. We will develop a policy for controlling the movement of trains in a railroad sytem and prove that the policy is sound, in the sense that it achieves a certain notion of safety. The soundness of the policy is proved completely automatically (in a fraction of a second), but we also provide a structured proof for it in Athena's natural deduction framework, which is then successfully 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 $P_1, \ldots, P_n$ as premises and attempt to derive the desired conclusion P

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. Usually 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

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

This is the problem of *abduction* [9, 8], which proceeds from the conclusion to premises.

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 succesful, use the information conveyed by that model to modify C appropriately and then loop back to step 2; if unsuccessful, go to step 3.

We illustrate this algorithm in Section 3. The individual steps of the algorithm are semi-mechanical (since 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: at every iteration through the loop, the simplest possible countermodel is produced, and this greatly facilitates the conjecture of a general condition that weeds 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 [7] case study by Daniel Jackson [6], which was in turn inspired by a presentation on modelling 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, February 2000. The Alloy formulation is based on Tarski's calculus of relations, whereas our formulation is given in conventional multi-sorted first-order logic and uses Athena.

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 may 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 mere collection of segments. We will capture this connectivity via 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.

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 shall 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 three axioms so far:

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

$$(\forall\, s)\, \texttt{overlaps}(s, s) \tag{3}$$

$$(\forall\, s_1, s_2)\, \texttt{overlaps}(s_1, s_2) \Rightarrow \texttt{overlaps}(s_2, s_1) \tag{4}$$

2.2 Capturing the state of the system

How do we formally capture a configuration ("snapshot") of the railroad system at a given point in time? In order to know the state of the system we need to know at least two things. First, the distribution of the trains amongst 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:

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

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,[1] 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 may now ask what would be an appropriate policy for controlling train motion that guarantees this safety criterion. We make this more precise as follows: we define a *sound safety policy* as a 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 the above sense.

3 Abduction via model generation

Initially we may well be at a loss in guessing what the appropriate constraints are. 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:

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

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

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

```
(define safety-statement
  (forall ?x ?y
    (if (and (reachableFrom ?y ?x)
             (safe ?x)
             true)
        (safe ?y))))
```

Predictably, perhaps, this statement isn't true, and when we try to prove it automatically by issuing the method call `(!prove safety-statement)`, we fail. (Refer to the appendix for a brief review of theorem proving and model generation in Athena.)

Now let us see *why* this does not hold. We will try to find a countermodel falsifying this policy, 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 safety-statement)                    (5)
```

This command attempts to find a model for the collection of all the propositions in the current assumption base *plus* the negation of `safety-theorem`. 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 `safety-theorem` 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 (5) results in the following output:

```
A model was found.

State has 2 elements: state-1, state-2.
Segment has 2 elements: segment-1, segment-2.
Train has 2 elements: train-1, train-2.

Press enter to see the function/relation definitions...
```

When the user presses enter, the extensions of the various functions and relations are presented as follows:

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

segOf(train-1, state-1) = segment-1
segOf(train-1, state-2) = segment-2
segOf(train-2, state-1) = segment-2
segOf(train-2, state-2) = segment-2

safe(state-1) = true
safe(state-2) = false
```

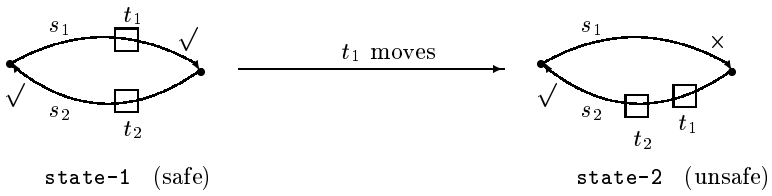


Fig. 1. A countermodel falsifying the trivial safety constraint *true*.

```
reachableFrom(state-1, state-1) = true
reachableFrom(state-1, state-2) = true
reachableFrom(state-2, state-1) = true
reachableFrom(state-2, state-2) = true

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

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

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, `safety-theorem` is false in this model. The reason for the failure becomes evident when we inspect the above output. 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 1. 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 `safety-statement` 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

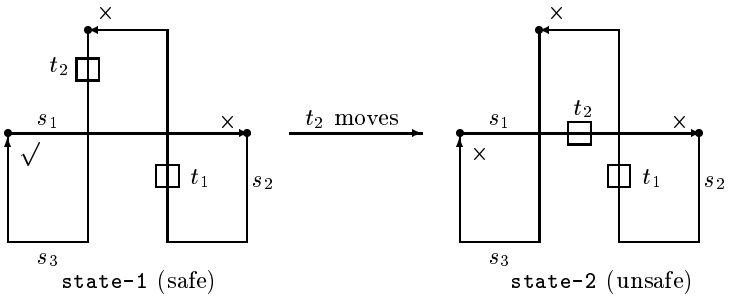


Fig. 2. A countermodel falsifying constraint C_1.

the command `(falsify safety-statement)` results in the output shown in Appendix C. 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 $\texttt{succ}(s_1, s_2)$, $\texttt{succ}(s_2, s_3)$, and $\texttt{succ}(s_3, s_1)$. 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. Afer examining the extensions of these relations and functions, we see that the first state is as depicted in the left half of Figure 2; namely, t_1 is on s_2, which has a closed gaten 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 2, 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.

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. Accordingly, we introduce a new constraint C_1':

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

and redefine `safety-statement` to be the proposition

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

Unfortunately, when we attempt to prove this latest version automatically,

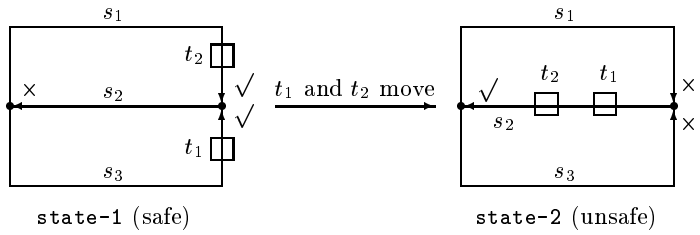


Fig. 3. A countermodel falsifying constraint C_1'.

we fail again. Returning to the model finder, we attempt to falsify this statement, which succeeds via the countermodel shown in Figure 3 (the textual presentation of this countermodel as given by Athena is shown in the appendix). 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:

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

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 not show a green light in two different directions. We now redefine `safety-theorem` 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 countermode shown in Figure 4 (we omit the textual presentation of the model for space reasons). The problem is essentially a generalization of the situtation depicted by the countermodel in Figure 3. 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:

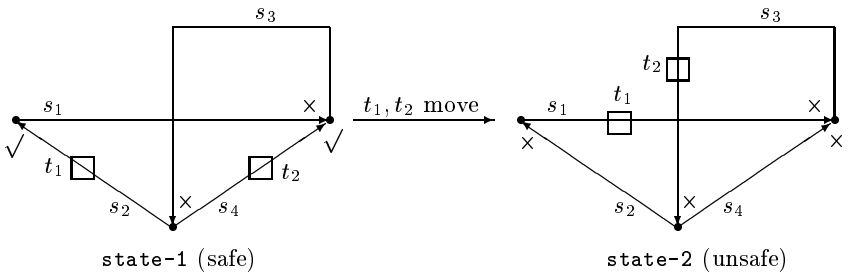


Fig. 4. A countermodel falsifying constraint C_2.

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

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

4 Automated theorem proving and proof representation

We have automatically verified that `safety-theorem` holds, and while that should boost our confidence in our policy, it is not quite good enough. As the engineers in charge of formulating a safety policy, 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 about a system should serve as documentation: it should explain why the system has this or that property. To that end, the proof should be structured, given in a "natural-deduction" style resembling informal mathematical reasoning, and at a high level of abstraction. Flat proofs in Hilbert-style or resolution systems do not meet these criteria.

Athena proofs are expressed in a block-structured ("Fitch-style" [14]) natural deduction calculus. High-level reasoning idioms that are frequently encountered in common mathematical discourse 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 reasoning and keeping the proof 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 parlance; see Section A of the Appendix.) 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 easier to write and remarkably useful in making proofs more modular and abstract. As this example will illustrate, writing tactics can pay dividends even in simple proofs.

In what follows we will 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' (i.e., $C_1'(x)$ and $C_2'(x)$ hold); then y is also safe.*

Proof. Pick arbitrary states x and y and assume that x is safe; 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

$$\texttt{overlaps}(\texttt{segOf}(t_1, y), \texttt{segOf}(t_2, y)) \tag{6}$$

We now ask: did either train move in the transition from state x to y, or did they both stay on the same segment? Clearly, 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 virtue of the law of the excluded middle. We will now show that in either of these two cases, a contradiction ensues.

Consider case_2 first, i.e., assume

$$\texttt{segOf}(t_1, y) = \texttt{segOf}(t_1, x) \tag{7}$$
$$\texttt{segOf}(t_2, y) = \texttt{segOf}(t_2, x) \tag{8}$$

Then, from (7), (8), and (6), we conclude

$$\texttt{overlaps}(\texttt{segOf}(t_1, x), \texttt{segOf}(t_2, x)) \tag{9}$$

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

$$\texttt{segOf}(t_1, y) \neq \texttt{segOf}(t_1, x) \tag{10}$$

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:

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

$$\neg\texttt{closed}(\texttt{segOf}(t_1, x), x) \tag{12}$$

We now perform a case analysis depending on whether or not t_2 moved as well. Clearly, there are only two cases: either it did move or it did not:

1. Suppose first that, like t_1, t_2 also moved, so that:

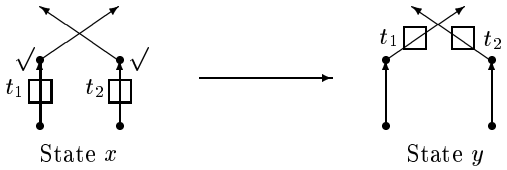
   $$\texttt{segOf}(t_2, y) \neq \texttt{segOf}(t_2, x) \tag{13}$$

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

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

   $$\neg\texttt{closed}(\texttt{segOf}(t_2, x), x) \tag{15}$$

   Hence, pictorially, the transition from x to y can be represented as follows:

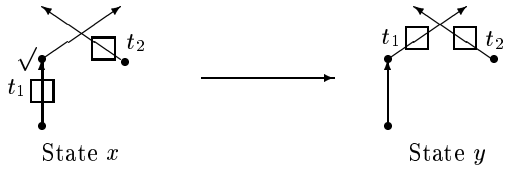
   But this means that the segments $\texttt{segOf}(t_1, x)$ and $\texttt{segOf}(t_2, x)$ are joinable: (a) they are distinct (if they were identical, then x would be unsafe, since $t_1 \neq t_2$ and `overlaps` is reflexive, contrary to our assumption); and (b) they have overlapping successors (from (6), (11), and (14)). Therefore, as shown in the left side of the above picture, 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.
2. By contrast, suppose that t_2 did not move during this state transition:

   $$\texttt{segOf}(t_2, y) = \texttt{segOf}(t_2, x) \tag{16}$$

   In that case, (6) entails

   $$\texttt{overlaps}(\texttt{segOf}(t_1, y), \texttt{segOf}(t_2, x)) \tag{17}$$

   Graphically, we have:



It is clear that this case violates constraint C_1' in state x: `segOf`(t_1, x), the segment of t_1 in x, is the predecessor of a segment that overlaps with an occupied segment, namely, `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. □

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 by 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 5, 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.

Observe that lexical scoping is important in giving free identifiers such as `hyp` within the body of `M` the appropriate denotation.

```
(safety-result BY
  (pick-any x y
    (assume-let ((hyp (and (safe x)
                           (reachableFrom y x)
                           (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 reachableFrom-definition hyp]))
                                   (P2 (!derive (not (closed (segOf r1 x) x))
                                                [hyp1 reachableFrom-definition hyp]))
                                   (c1 (assume-let ((case1 (not (= (segOf r2 y) (segOf r2 x)))))
                                         (dlet ((P3 (!derive (succ (segOf r2 x) (segOf r2 y))
                                                             [case1 reachableFrom-definition hyp]))
                                                (P4 (!derive (not (closed (segOf r2 x) x))
                                                             [case1 reachableFrom-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 [C2A-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 C1A-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. 5. Athena proof of the safety result

References

1. K. Arkoudas. Denotational Proof Languages. PhD dissertation, MIT, 2000.
2. K. Claessen and N. Sorensson. New techniques that improve Mace-style finite model building. In *Model Computation—principles, algorithms, applications*, Miami, Florida, USA, 1973.
3. 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.
4. 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.
5. 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.
6. D. Jackson. Railway Safety. `http://alloy.mit.edu/case-studies.html`, 2002.
7. Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.
8. J.R. Josephson and S.G. Josephson, editors. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge University Press, 1994.
9. 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.
10. L. Lamport. How to write a proof. Research Report 94, Systems Research Center, DEC, February 1993.
11. Z. Manna and R. Waldinger. *The logical basis for computer programming*. Addison Wesley, 1985.
12. W. McCune. A Davis-Putnam program and its application to finite first-order model search. Technical Report ANL/MCS-TM-194, ANL, 1994.
13. L. Paulson. *Isabelle, A Generic Theorem Prover*. Lecture Notes in Computer Science. Springer-Verlag, 1994.
14. F. J. Pelletier. A Brief History of Natural Deduction. *History and Philosophy of Logic*, 20:1–31, 1999.
15. A. Voronkov. The anatomy of Vampire: implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2), 1995.
16. C. Weidenbach. Combining superposition, sorts, and splitting. volume 2 of *Handbook of Automated Reasoning*. North-Holland, 2001.

A Athena primer

Athena maintains a global set of propositions called the *assumption base*. We can think of the elements of the assumption base as our premises—propositions that we regard as true. Initially the system starts with the empty assumption base. Every time an axiom is asserted or a theorem is proved or a definition of a new symbol is introduced, the corresponding proposition is inserted into the assumption base. We can list the elements of the assumption base at any point in time with the nullary function `get-assumption-base`, which returns a list of all and only the propositions that are in the current assumption base.

Models are generated by the unary function `find-model`, which takes a list of propositions $P_1, \ldots, P_n$ and attempts to build a finite model that satisfies every P_i. The unary function `falsify` attempts to invalidate a given proposition P in the context of the current assumption base. That is, `(falsify` P`)` attemtps to build a model that satisfies every proposition in the assumption base but falsifies P. Accordingly, `(falsify` P`)` can be understood as a shorthand for `(find-model (add (not` P`) (get-assumption-base)))`, where `add` is a "consing" function that prepends a given element to a given list.

Inference in Athena is performed mainly through the application of *methods*. A method can be thought of as a special type of function. There are two main differences between methods and "normal" functions. First, methods only return propositions, whereas a regular function can return a number or a string or a record or a file, or indeed any data value of any type. Secondly, if and when a method returns a proposition, that proposition is guaranteed to be a logical consequence of the assumption base. This soundness property is ensured by the formal semantics of Athena. So methods are proposition-producing functions that are constrained to play inside a sandbox, so to speak, in such a way that they can only generate results that follow logically from the assumption base.

Syntactically, method applications are of the form `(!`$E\ E_1 \cdots E_n$`)`. The exclamation mark indicates that E is a method. $E_1 \cdots E_n$ are the arguments to that method. By contrast, the general form of a function call is `(`$E\ E_1 \cdots E_n$`)`, with no exclamation mark. Scheme and other Lisp variants have the same syntax for function calls.

Athena has various built-in primitive methods, mostly introduction and elimination rules for the various logical connectives. For instance, `left-and` is a unary method that detaches the left component of a conjunction. Suppose, for instance, that some conjunction `(and P Q)` is in the current assumption base. Then the method call `(!left-and (and P Q))` will result in the proposition P. A corresponding method `right-and` would produce `Q`. Note that `(and P Q)` *must* be in the assumption base for the method call `(!left-and (and P Q))` to succeed; otherwise we will get an error message. This is to ensure soundness. Soundness in Athena means that if a proof derives some conclusion P in assumption base β, then P must follow logically from β. If we don't demand that the argument supplied to `left-and` be in the assumption base, soundness would be lost. For instance, the application `(!left-and (and false true))` would derive the conclusion `false` from the empty assumption base.

The binary method `both` is an introduction rule for conjunctions. The call `(!both P Q)` results in the conclusion `(and P Q)`, provided that both `P` and `Q` are in the assumption base (it is an error if either of them is not). The binary method `uspec` is an elimination rule for universal generalizations. For example, let `P` be the proposition `(forall ?x (isMale (father ?x)))`. If `P` is in the assumption base, then `(!uspec P joe)` will result in the conclusion `(isMale (father joe))`.

Multiple inference steps can be put together with `dbegin` ("deductive begin"). For any two proofs D_1 and D_2, the construction `(dbegin` $D_1\ D_2$`)` is a new composite proof that performs D_1 and D_2 sequentially. First, D_1 is evaluated in the current assumption base β, producing some conclusion P_1. Then P_1 *is added*

to the assumption base and we continue with the second proof D_2. Thus D_2 is evaluated in $\beta \cup \{P_1\}$. The result of D_2 becomes the result of the entire `dbegin`. This allows for lemma formation: the conclusion of D_1 (P_1) serves as a lemma inside D_2. More than two deductions can appear inside a `dbegin`. For $n > 2$, the semantics of `(dbegin` $D_1 \cdots D_n$`)` are given by desugaring to the $n = 2$ case. For instance, `(dbegin` D_1 D_2 D_3`)` is defined as `(dbegin` D_1 `(dbegin` D_2 D_3`))`. For instance, suppose that `(and P Q)` is in the assumption base. Then the following proof will derive the conclusion `(and Q P)` (a semicolon `;` starts a comment that extends to the end of the line):

```
(dbegin (!left-and  (and P Q))   ; this will derive P
        (!right-and (and P Q))   ; this will get Q
        (!both Q P))             ; and finally, this will get (and Q P)
```

The `BY` construct can enhance proof readability. Any proof D that produces a conclusion P can also be written as `(`P `BY` D`)`. This form uses P as an annotation: it declares that the result of D is P. Accordingly, to evaluate `(P BY` D`)` we first evaluate D and then check to make sure that the result is P. If it is, we return it as the result; if it is not, we raise an error. For instance, the above proof can also be written as:

```
(dbegin (P BY (!left-and  (and P Q)))
        (Q BY (!right-and (and P Q)))
        ((and Q P) BY (!both Q P)))
```

Note that the comments are no longer necessary.

A more flexible version of `dbegin` is `dlet`. The proof `(dlet ((`I D_1`)) `D_2`)` is similar to `(dbegin` D_1 D_2`)`: first we evaluate D_1, then we add its conclusion to the assumption base and proceed to evaluate D_2; the conclusion of the D_2 becomes the conclusion of the entire `dlet`. The difference is that within D_2, the identifier I will refer to the conclusion of D_1. So `dlet` performs naming in addition to proof composition. Here too we can use desugaring to generalize this construct to multiple pairs of identifier-proofs: `(dlet ((`I_1 D_1`)` $\cdots$ `(`I_n D_n`)) `D`)`. For instance, `(dlet ((`I_1 D_1`) (`I_2 D_2`)) `D`)` is defined as

`(dlet ((`I_1 D_1`)) (dlet ((`I_2 D_2`)) `D`))`.

Using `dlet` we can express our earlier example as:

```
(dlet ((left-part  (!left-and (and P Q)))
       (right-part (!right-and (and P Q))))
  (!both right-part left-part))
```

Finally we come to *hypothetical* (or "conditional") proofs. Hypothetical proofs are used to establish conditionals of the form $P \Rightarrow Q$. They work as follows: we take P as a working assumption and proceed to derive Q; if and when we are done, we discharge P and conclude $P \Rightarrow Q$. To take a simple example, consider the tautology $(P \wedge Q) \Rightarrow (Q \wedge P)$. In traditional Fitch style, a proof of this tautology would be of the following form:

1.	$P \wedge Q$	
2.	P	1, left $\wedge$-elimination
3.	Q	1, right $\wedge$-elimination
4.	$Q \wedge P$	3, 2, $\wedge$-introduction
5.	$P \wedge Q \Rightarrow Q \wedge P$	1—4, $\Rightarrow$-introduction

Lines 2 through 4 are within the *scope* of the assumption $P \wedge Q$ in line 1 and constitute a *subordinate proof*. The notion of *assumption scope* (and corresponding nested subordinate proofs) is the hallmark of natural deduction. In Fitch systems the scope of an assumption A is indicated graphically by a perpendicular line extending from A all the way down to the end of the subordinate proof; a horizontal line is also drawn underneath A. Subordinate proofs within the scope of an assumption A are free to use A as given. Note that line 5 *discharges* the assumption of line 1, pulling the scope back out one level, and derives the conditional $P \wedge Q \Rightarrow Q \wedge P$.

In Athena, hypothetical proofs are of the form `(assume` A D`)`. Here A is the hypothesis and D is the subordinate proof that represents the scope of A. We also refer to D as the *body* of the hypothetical proof. The semantics of such proofs are simple: to evaluate `(assume` A D`)` in an assumption base β, we add A to the assumption base and evaluate the body D in the augmented assumption base $\beta \cup \{A\}$; if and when the evaluation of D produces a conclusion B, we return the conditional $A \Rightarrow B$ as the final result.

For instance, the conditional `(if (and P Q) (and Q P))` can be derived by the following proof (in *any* assumption base):

```
(assume (and P Q)
  (dbegin (!left-and  (and P Q))
          (!right-and (and P Q))
          (!both Q P)))
```

The hypothesis of this `assume` is `(and P Q)`. The body of the `assume` is the `dbegin` deduction, which constitutes the scope of the hypothesis `(and P Q)`. Therefore, the proposition `(and P Q)` can be freely used within that scope as a premise (e.g., supplied as an argument to `left-and`). Note that there is no need to explictly discharge the hypothesis. Athena will do the discharge automatically when it has finished evaluating the `assume`.

Oftentimes the hypothesis of a conditional deduction is a large proposition, so we would rather give it a short name and refer to it by that. We can do this with a `dlet`, e.g.:

$$\texttt{(dlet ((}I\ \cdots\texttt{)) (assume }I\ D\texttt{))} \tag{18}$$

where $\cdots$ is the hypothesis to be named I. This is such a convenient construct that Athena has a built-in mechanism for it: `(assume-let ((`$I\ \cdots$`)) `D`)`. This not only postulates the hypothesis $\cdots$ but also gives it the name I, by which the body D can then refer to it. So this `assume-let` is just syntax sugar for (18).

When we inspect the above deduction it becomes clear that the particular identities of `P` and `Q` are immaterial. The proof will work properly no matter what

propositions `P` and `Q` are. This suggests the possibility of parametric *abstraction*. Just as we can abstract a computation such as `(* a a)` over `a` and formulate a squaring procedure `(lambda (a) (* a a))` which can then be applied to infinitely many arguments, we are likewise able to abstract a proof D over one or more parameters $I_1 \cdots I_n$ and formulate a method `(method (`$I_1 \cdots I_n$`) D)` which can then be applied to any given arguments as if it were a primitive built-in method. For instance, we can abstract the foregoing hypothetical proof into a general method named, say, `commute-and`:

```
(define commute-and
  (method (P Q)
    (assume (and P Q)
      (dbegin (!left-and  (and P Q))
              (!right-and (and P Q))
              (!both Q P)))))
```

We can now apply this method to any arguments. E.g., `(!commute-and true false)` will derive the conditional `(if (and true false) (and false true))`.

A.1 Reasoning with quantifiers

Athena has one introduction and one elimination mechanism for each of the two quantifiers, so there are four mechanisms altogether for quantifier reasoning. Two of them (`uspec` and `egen`) are primitive methods; the other two are native syntax forms. We describe them below.

Universal quantifier introduction Universally quantified propositions are derived with deductions of the form `(pick-any` I D`)`. These are meant to model the way in which human mathematicians establish universal generalizations. For instance, to prove that every prime integer is odd, we might say

Consider any integer n and suppose that n is prime. Then D. (19)

where D is some deduction with free occurrences of n which establishes that n is odd. It is essential here that the name n should denote an arbitrary object, i.e.,

```
(assume-let ((hyp (and (P x) (Q x))))
  (pick-any y
    (dlet ((lemma ((and (P y) (Q y)) BY (!uspec hyp y))))
      ((P y) BY (!left-and lemma)))))
```

Universal quantifier elimination Use the primitive method `uspec` ("universal specialization")

Existential quantifier introduction Use the primitive method `egen` ("existential generalization")

Existential quantifier elimination Use a deduction of the form

(pick-witness I E D)

A.2 Automated reasoning

Automated theorem proving in Athena is done via the primitive binary method `derive`, which takes a proposition P (the goal) and a list of premises $P_1, \ldots, P_n$. The call (!derive P [$P_1 \cdots P_n$]) will attempt to prove the goal P from the premises $P_1, \ldots, P_n$, provided that the latter are in the current assumption base (if they are not, an error will be reported). If this succeds within a preset time limit (60 seconds by default), P is returned as a theorem; otherwise the call fails.

The unary method `prove` attempts to derive a goal P from the entire assumption base. Therefore, (!prove P) can be viewed as an abbreviation for the call (!derive P (get-assumption-base)).

B Athena formalization of the railroad model

The Athena model of the railroad system is presented in its entirety in Figure 6.

C Second countermodel

The following is Athena's verbatim presentation of the countermodel depicted in Figure 2:

```
A model was found.

State has 2 elements: state-1, state-2.
Segment has 3 elements: segment-1, segment-2, segment-3.
Train has 2 elements: train-1, train-2.

Press enter to see the function/relation definitions...

succ(segment-1, segment-1) = false
succ(segment-1, segment-2) = false
succ(segment-1, segment-3) = true
succ(segment-2, segment-1) = true
succ(segment-2, segment-2) = false
succ(segment-2, segment-3) = false
succ(segment-3, segment-1) = false
succ(segment-3, segment-2) = true
succ(segment-3, segment-3) = false

segOf(train-1, state-1) = segment-1
segOf(train-1, state-2) = segment-1
segOf(train-2, state-1) = segment-3
segOf(train-2, state-2) = segment-2
```

```
(load-file "util.ath")

(domains Train Segment State)

(declare segOf (-> (Train State) Segment))

(declare succ overlaps (-> (Segment Segment) Boolean))

(declare closed (-> (Segment State) Boolean))

(assert (reflexive overlaps) (symmetric overlaps) (intransitive succ))

(define-symbol occupied
  (forall ?s ?x
    (iff (occupied ?s ?x)
         (exists ?t (= (segOf ?t ?x) ?s)))))

(define-symbol safe
  (forall ?x
    (iff (safe ?x)
         (forall ?t1 ?t2
           (if (not (= ?t1 ?t2))
               (not (overlaps (segOf ?t1 ?x) (segOf ?t2 ?x))))))))

(define-symbol reachableFrom
  (forall ?x ?y
    (iff (reachableFrom ?y ?x)
         (forall ?t
           (if (not (= (segOf ?t ?x) (segOf ?t ?y)))
               (and (succ (segOf ?t ?x) (segOf ?t ?y))
                    (not (closed (segOf ?t ?x) ?x))))))))

(define-symbol joinable
  (forall ?s1 ?s2
    (iff (joinable ?s1 ?s2)
         (and (not (= ?s1 ?s2))
              (exists ?s1' ?s2'
                (and (overlaps ?s1' ?s2')
                     (succ ?s1 ?s1')
                     (succ ?s2 ?s2'))))))))


(define-symbol C1'
  (forall ?x
    (iff (C1A ?x)
         (forall ?s1 ?s2 ?s3
           (if (and (succ ?s1 ?s2)
                    (overlaps ?s2 ?s3)
                    (occupied ?s3 ?x))
               (closed ?s1 ?x))))))


(define-symbol C2'
  (forall ?x
    (iff (C2A ?x)
         (forall ?s1 ?s2
           (if (and (joinable ?s1 ?s2)
                    (not (closed ?s1 ?x)))
               (closed ?s2 ?x))))))
```

Fig. 6. The railroad model in Athena.

```
safe(state-1) = true
safe(state-2) = false

reachableFrom(state-1, state-1) = true
reachableFrom(state-1, state-2) = false
reachableFrom(state-2, state-1) = true
reachableFrom(state-2, state-2) = true

overlaps(segment-1, segment-1) = true
overlaps(segment-1, segment-2) = true
overlaps(segment-1, segment-3) = false
overlaps(segment-2, segment-1) = true
overlaps(segment-2, segment-2) = true
overlaps(segment-2, segment-3) = true
overlaps(segment-3, segment-1) = false
overlaps(segment-3, segment-2) = true
overlaps(segment-3, segment-3) = true

closed(segment-1, state-1) = true
closed(segment-1, state-2) = true
closed(segment-2, state-1) = true
closed(segment-2, state-2) = true
closed(segment-3, state-1) = false
closed(segment-3, state-2) = true
```

D Third countermodel

The following is Athena's verbatim presentation of the countermodel depicted in Figure 3:

```
succ(segment-1, segment-1) = false
succ(segment-1, segment-2) = true
succ(segment-1, segment-3) = false
succ(segment-2, segment-1) = true
succ(segment-2, segment-2) = false
succ(segment-2, segment-3) = true
succ(segment-3, segment-1) = false
succ(segment-3, segment-2) = true
succ(segment-3, segment-3) = false

segOf(train-1, state-1) = segment-3
segOf(train-1, state-2) = segment-2
segOf(train-2, state-1) = segment-1
segOf(train-2, state-2) = segment-2

safe(state-1) = true
safe(state-2) = false
```

```
reachableFrom(state-1, state-1) = true
reachableFrom(state-1, state-2) = true
reachableFrom(state-2, state-1) = true
reachableFrom(state-2, state-2) = true

overlaps(segment-1, segment-1) = true
overlaps(segment-1, segment-2) = false
overlaps(segment-1, segment-3) = false
overlaps(segment-2, segment-1) = false
overlaps(segment-2, segment-2) = true
overlaps(segment-2, segment-3) = false
overlaps(segment-3, segment-1) = false
overlaps(segment-3, segment-2) = false
overlaps(segment-3, segment-3) = true

closed(segment-1, state-1) = false
closed(segment-1, state-2) = true
closed(segment-2, state-1) = true
closed(segment-2, state-2) = false
closed(segment-3, state-1) = false
closed(segment-3, state-2) = true
```