Prototyping Formal Methods Tools: A Protocol Analysis Case Study

Abigail Siegel, Mia Santomauro, Tristan Dyer,
Tim Nelson, and Shriram Krishnamurthi

Computer Science Department, Brown University, Providence, RI, USA

Abstract. Modern-day formal methods tools are more than just a core solver: they also need convenient languages, useful editors, usable visualizations, and often also scriptability. These are required to attract a community of users, to put ideas to work in practice, and to conduct evaluations of the formalisms and core technical ideas. Off-the-shelf solvers address one of these issues but not the others. How can full prototype environments be obtained quickly?
We have built Forge, a system for prototyping such environments. In this paper, we present a case-study to assess the utility of Forge. Concretely, we use Forge to build a basic protocol analyzer, inspired by the Cryptographic Protocol Shape Analyzer (CPSA). We show that we can obtain editing, basic visualization, and scriptability at no extra cost beyond embedding in Forge, and a modern, domain-specific visualization for relatively little extra effort.

1 Introduction

Formal methods are (finally) surging in popularity, including numerous domain-specific tools, even of industrial origin [2, 9, 18, 8, 7, 41, 46]. This suggests that there are many new tools that people might want to build; as exposure grows, so will the number of tools. But there's a long road from a formalism to a tool. Researchers need to quickly build prototypes that can be experimented with and refined (and perhaps even turned into a bespoke tool).

Many tools [36, 29, 32, 5, 33, 38, 28, 44] already layer a domain atop a model-finder like Alloy [23], Alloy's core engine Kodkod [54], SMT, or SAT. While these are wonderful as embedded solvers, they are only the beginning, not the end, to building a *useful* tool. Some additional concerns include:

- the development environment (IDE) experience;
- translating the domain-specific surface input language;
- visualization of the output;
- perhaps even domain-specific and interactive output visualization; and
- extensibility and scriptability.

These concerns are not academic. Tools benefit from user communities. While early adopters will use almost any interface, as communities grow, they expect all the standard modern conveniences.

Independent of community growth, our formalisms benefit from (and need) evaluations with users, especially because these can cough up unpleasant surprises [11]. But to perform such evaluations, we must equip them with at least minimal usable interfaces. Otherwise, our studies will be studying the (poverty of the) interface, not the formalism and its consequences.

Our response to this problem is a new framework, Forge[1] (the name is a tribute to Alloy), that enables researchers to quickly prototype tools. Forge is based on the *language-oriented programming* (LOP) principle [16] of the Racket programming system. That is, Racket is a system (and language) designed for building (programming) languages. The resulting languages can be used from the DrRacket programming environment [17] or with external environments (such as Visual Studio Code) using the Common Language Interface. Forge provides the Kodkod [54] and Pardinus [10] solvers. It also incorporates the Sterling [15] visualizer, which enables domain-specific visualizations. Finally, domain-specific programs can be scripted using the Racket language.

This paper presents a case study that exercises these aspects of Forge. Concretely, we will use Forge to build a prototype analyzer for cryptographic protocols, inspired by Joshua Guttman's Cryptographic Protocol Shapes Analyzer (CPSA) [12]. This prototype was largely executed by a pair of undergraduates (the first two authors) as part of a *course project* (while taking a regular course load). We believe that this demonstrates the potential utility of frameworks like Forge, and hope that this work prompts further development to support *end-to-end* prototyping of formal tools.

2 End-To-End Language-Oriented Modeling

The key philosophy of Forge lies in extending the idea of LOP to language-oriented *modeling*. We illustrate this process in Figure 1, which is organized by tiers according to the different user perspectives involved. For concreteness, we specialize the presentation to our specific crypto-analysis case study.

Atop the pre-existing Forge engine, *Tool Authors* (in this case study, the undergraduate lead authors) model their domain (the "Base spec"), define domain-specific languages (DSLs) in Racket, the translation of those languages to Forge constraints (`#lang`), and—if needed—a domain-specific visualizer ("Custom Visualization"). These enable other user perspectives: domain experts, such as *Protocol Creators*, use DSLs (like CPSA's `defprotocol` syntax) to specify artifacts of interest in their domain without needing expertise in relational logic. *Analysts* can then phrase queries about protocol behavior. They may use Forge's query language, a DSL (such as CPSA's `defskeleton` syntax), or both, and benefit from domain-customized output. Others, such as students, might even bypass the DSLs entirely and only interact with visualizations produced by others. While a specific user may naturally belong to multiple tiers (e.g., a protocol creator may wish to double-check their specification by viewing example executions), we find this separation a useful way to think about different tool perspectives.

[1] Available at: www.forge-fm.org

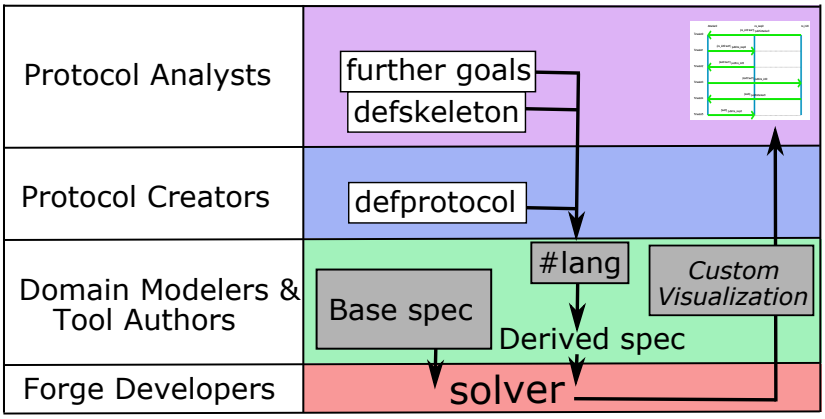


Fig. 1: Tiered organization of Domain-Specific Modeling in Forge. Components that must be implemented for each new domain (e.g., our case-study prototype) are shaded in grey. A base specification for the domain is enriched with additional constraints generated from the domain-specific input. Results from the solver are relayed to a custom visualizer. As the process is embedded in Racket, additional structure can be added via scripting (not shown) from outside the core workflow.

Crucially, this process is not specific to cryptographic protocols. Any domain that can be modeled in the relational logic of Forge (which it shares with Alloy) is a potential target of this approach.

We now briefly step through the perspective of each user, and address corresponding system-design concerns raised in Section 1. As a running example throughout, we use the Needham-Schroeder [35] asymmetric protocol with the known (Lowe [26]) vulnerability, taken verbatim from the CPSA example repository (https://github.com/mitre/cpsa). Concretely, this protocol describes a three-step secret exchange between two principals (`init`iator and `resp`onder) facilitated by a public-key cryptosystem.

2.1 Protocol Analysts: Custom Visualization and Queries

A concrete example of Needham-Schroeder in action might look like Figure 2, where horizontal arrows denote the flow of messages between principals. Our case-study prototype's model and visualization are based on the strand-space formalism [51], just as is CPSA. We discuss differences in logic (Section 4) and visualization (Section 6) later, but note that our visual layout concretizes the Dolev-Yao [13] perspective: the attacker is synonymous with the medium of communication. Our visualizer is also interactive: users can click at any point on the diagram to see the state of agents' knowledge at that time. Figure 3 shows one such report: the initiator's knowledge before the first message is sent.

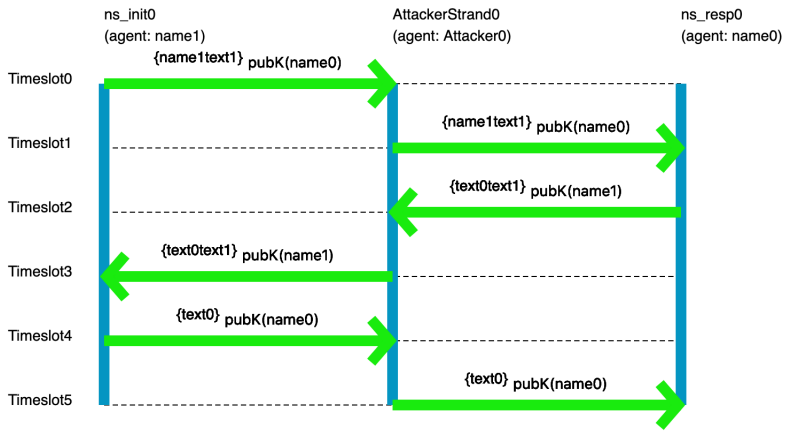


Fig. 2: A good execution of Needham-Schroeder. The "Attacker" strand represents the medium of communication. As the secrets (`text0` and `text1`) are encrypted, they are learned (in this execution) by only the initiator and responder. The "agent:" annotations denote which agent owns each strand.

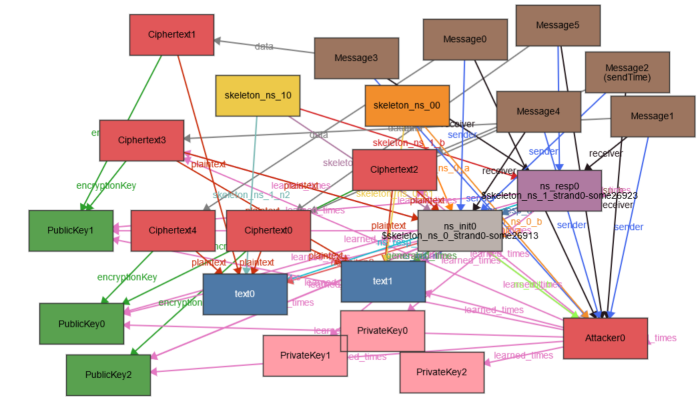
Visual Design Considerations There are many different visualizations that a tool author might create. The lowest-cost approach would be to use the standard Alloy-style directed graphs that Forge provides by default. However, these fail to communicate domain-specific intuitions, expose unnecessary modeling details, and can become overwhelming after a certain level of complexity is reached. Instead, Forge leverages the Sterling [15] visualizer, which allows tool authors to build their own custom visualizations in JavaScript. These then execute in the browser, and benefit from the many advantages of a modern web interface.

Fig. 3: Detail of initiator's knowledge in the first timestep.

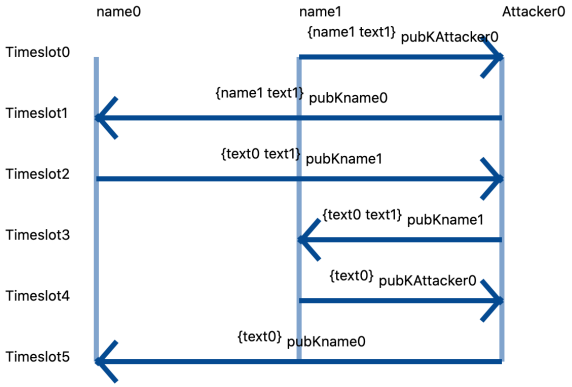
Figure 4 shows three points in the design space of visualizing an *attack on*, rather than a good run of, Needham-Schroeder:

1. the default Alloy-style visualization, projected by timeslot and with unused atoms removed;
2. a lightweight (roughly 100 lines of JavaScript) custom visualization; and
3. the full interactive visualizer (900 lines of JavaScript).

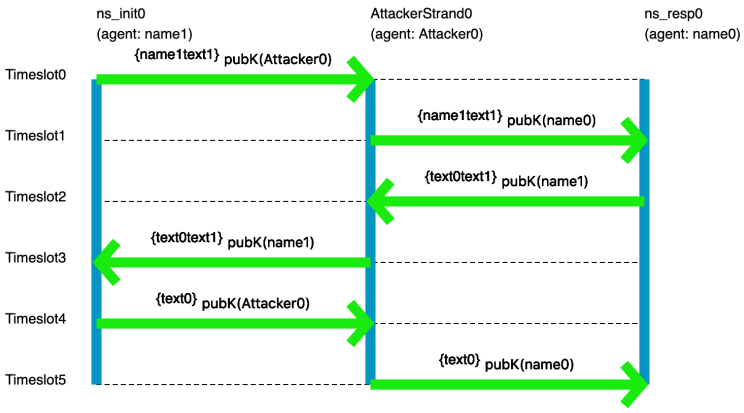
Note that even with some effort to clean up the display, the default visualization is cluttered with modeling details and can be difficult to break down. While Alloy and Sterling do provide an alternative table-based modality, these same issues apply (with, e.g., 38 rows in the `learned_times` relation alone). In contrast, the custom visualizations at least communicate some pertinent information at a brief glance, and the full visualization provides more (e.g., agent knowledge) on click events. Section 6 presents the visualizer in greater detail.



(a) Default Visualization (for one timeslot)



(b) Lightweight Custom Visualization



(c) Full Custom Visualization

Fig. 4: Three different visualizations of a concrete attack on Needham-Schroeder. Here, the man-in-the-middle attack is realized by the initiator starting a session with the attacker or an agent whose private key is known to the attacker.

Development Environment Building atop Racket gives us immediate recourse to the DrRacket IDE, which comes with useful features such as error highlighting, a Read-Eval-Print-Loop (REPL), debugging features, etc. Moreover, many other mature editors (such as Visual Studio Code) are readily usable via a language server interface.

Query Language Forge exposes a parenthetical query language based on the first-order relational logic of Kodkod [54]. Generating the known *attack* on Needham-Schroeder (instead of a good execution) requires only asking for scenarios where both of the initiator's nonce values are eventually learned by the attacker:

```
(in (+ (join ns_init ns_init_n1)
       (join ns_init ns_init_n2))
  (join Attacker learned_times Timeslot)))
```

Section 5 gives more detailed background for the identifiers used here. For now, we observe that `ns_init` represents an initiator strand and that `ns_init_n1` and `ns_init_n2` are relations that contain the values of each nonce variable for each initiator. The `learned_times` relation represents the state of each agent's knowledge at each point in time. The values for specific strands are obtained via relational `join` (i.e., lookup), as is the attacker's knowledge across all timeslots (the `Timeslot` relation). `+` denotes union, and `in` the (possibly improper) subset relationship.

2.2 The Protocol Creator: Translating Domain-Specific Input

In CPSA's input language, the Needham-Schroeder protocol is represented by:

```
(defprotocol ns basic
  (defrole init
    (vars (a b name) (n1 n2 text))
    (trace
     (send (enc n1 a (pubk b)))
     (recv (enc n1 n2 (pubk a)))
     (send (enc n2 (pubk b)))))
  (defrole resp
    (vars (b a name) (n2 n1 text))
    (trace
     (recv (enc n1 a (pubk b)))
     (send (enc n1 n2 (pubk a)))
     (recv (enc n2 (pubk b)))))
  (comment "Needham-Schroeder"))
```

The `defprotocol` construct specifies the behaviors corresponding to well-behaved protocol participants, and is the way a protocol author would describe their protocol in CPSA's domain perspective.

CPSA also provides a `defskeleton` construct, which describes fragments of execution that analysts use to search for specific protocol behaviors. For example, the Needham-Schroeder file from CPSA's example suite contains this skeleton:

```
(defskeleton ns
  (vars (a b name) (n2 text))
  (defstrand resp 3 (a a) (b b) (n2 n2))
  (non-orig (privk a) (privk b))
  (uniq-orig n2)
  (comment "Responder point-of-view"))
```

which defines a particular search in the space of executions. Each `defstrand` defines a single process executing the appropriate protocol role. The 2-tuples (e.g. `(n1 n1)`) are called *maplets* in CPSA parlance, and bind values (skeleton variables) to role variables in the protocol. The `non-orig` and `uniq-orig` annotations give constraints on how principals may behave. They enforce that these values are freshly chosen and either never sent by a principal in decryptable form (`non-orig`), or that their appearance originates on a single strand (`uniq-orig`).

One advantage of CPSA's parenthetical language is that no parser is required to process it; Racket macros can expand protocol and skeleton definitions directly into Forge formulas. While Racket permits non-parenthetical syntaxes [16], this underlying parenthetical layer saves domain modelers of having to construct source by unwieldy and bug-inducing string concatenation, as often happens when mapping to other tools. We trust that the Verified LISP [21] instantiation of Joshua Guttman would especially appreciate this.

2.3 Scripting and Extensibility

Users at any level may wish to script analysis in Forge for their own purposes. They might wish to numerate protocol runs for pattern mining, generate "ensembles" of differing runs, work with the solver iteratively [33], etc.

The core of Forge is implemented as a library in Racket. Users may work with the logic language directly (akin to what Alloy's UI provides), or use Forge as a library in a larger program. While Forge is meant to be used for prototyping "solver-aided" languages, it differs from tools like Rosette [52] by sharply separating the logic language from Racket. Thus, the engine need not be able to reason about (e.g.) recursion or other programming constructs, although computation over the logic language can still be scripted. The formula derived from a CPSA `defprotocol` s-expression can be used either as a helper predicate in the logic language or as a programmatic object in Racket. Likewise, the custom visualization applies to both naive scenario enumeration and custom exploration strategies [37, 27, 45].

Roadmap After some brief background (Section 3), this paper covers the technical heart of the prototype: the core model (Section 4), the translation from CPSA inputs to supplemental constraints (Section 5), and custom visualization (Section 6), which includes graphical demonstrations on further example protocols. We then examine performance (Section 7), summarize related work (Section 8) and conclude with a discussion of lessons learned (Section 9).

3 Relational Model Finding

Model-finding tools find concrete solutions that satisfy a given set of declarative constraints. *Relational* model-finders, of which Alloy [23] is an especially popular example, take input in a relational constraint language and produce relational structures as output. Alloy's core engine, Kodkod [54], translates input constraints into boolean logic and then invokes an off-the-shelf SAT-solver. There are various enhancements to Kodkod, such as Pardinus [10], which Forge uses directly. However, since all these derive from Kodkod, we will disambiguate by using it as our exemplar when we speak of Forge's solver engine.

We borrow from Milicevic [33] and others by calling the input to Kodkod a *specification*, rather than the broader Alloy community's use of "model". Using "model" in this way would conflict with the fact that, in a mathematical context, the term describes an interpretation for a (relational) language—which is the type of a model-finder's *output*, not its input.

4 Modeling Protocol Executions

Our base specification provides a generic framework into which individual protocols and skeletons may be instantiated. This common framework defines the notion of message passing between strands, the knowledge of various agents involved in protocol execution, the construction of ciphertext terms, and many other ideas central to approximating the strand-space perspective.

The sorts and subsorting relationships in our specification largely echo the basic CPSA algebra: a top-level `mesg` sort for arbitrary values, `skey` and `akey` sorts for symmetric and asymmetric keys, `text` for plain values like nonces, etc. An ordered `Timeslot` sort serves as an index for message events. Relations on these sorts track key ownership (`owners`, `pairs`), long-term keys (`ltks`), message contents (`data`), the state of each agent's knowledge at any given time (`learned_times`), ciphertext contents (`plaintext`), and other essential properties of a protocol run.

Using this relational signature, the specification imposes well-formedness criteria that should hold regardless of the specific protocol being examined. Briefly, these constraints include:

- standard type constraints (e.g., that every `Ciphertext` has exactly one encryption key);
- every `mesg` is a `key`, `name`, `text`, or `Ciphertext`;
- all messages are either sent to, or received from, the attacker strand;
- sent messages only include values known to the sender;
- the plaintext relation is acyclic;
- the `pairs` relation defines one unique key pair per `name`;
- the `ltks` relation defines a partial function on ordered pairs of `name`s; and
- a characterization of when an agent learns a value (the contents of a message they can encrypt, a value they have just generated, their own name, etc.)

This specification approximates the strand space formalism, but in the spirit of CPSA itself, it is worth examining the explicit and implicit assumptions made and briefly discussing their consequences. Indeed, it is worth noting that Forge's bounded relational logic was not always the most natural idiom to express our goals—we return to this in Section 9.

Concrete agents One of our goals was to explicitly represent the state of each agent's knowledge throughout a protocol execution. However, in general an agent may run multiple strands of the same protocol, and so our specification separates the notion of `strand` (and its variable bindings) from the corresponding agent (and its pool of knowledge at any given time). We make this explicit in our visualization (Section 6) by naming every strand's corresponding principal.

An explicit attacker The `Attacker` strand is synonymous with an untrusted communication medium and is thus an explicit version of the Dolev-Yao [13] adversary. We found this to be useful, both for debugging the prototype and for visualization, since it makes it easy to track exactly which messages are delayed or rewritten and what knowledge has been exposed.

Strands and Messages A satisfying model for our specification contains a set of strands, along with information about message send and receive events. Messages may involve an arbitrary (user-bounded) number of nested encryptions. We make two simplifying assumptions. First, we do not view strands as having a length, but rather a specific pattern of send and receive events over the duration of the run. No model can contain a partial strand. Second, message components are implicitly unordered. These choices are a semantic mismatch versus CPSA—and indeed prevent detection of some attacks!—but eased first-cut development, sufficed for the examples in Section 7, and could be corrected via standard techniques with some engineering effort.

Origination and Pre-existing Knowledge In CPSA, a strand *originates* a term if (broadly) that strand sends the term, and all other strands that send the term must first receive it. CPSA uses this idea to speak of a nonce being freshly created or a key never being sent by any honest participant. We echo this idea as:

```
pred originates[s: strand, d: mesg] { -- d originates on s
  some m: sender.s | { -- m sent by strand s
    d in subterm[m.data] -- contains d as a sub-term
    all m2: (sender.s + receiver.s) - m | { -- all else
      {m2 in m.^(~(next))} -- if m2 occurred before m
      implies
      {d not in subterm[m2.data]} -- d is not in m2
    }
  }
}
```

Moreover, since our specification explicitly represents knowledge, certain terms must originally come to be in an agent's knowledge-base. We enforce the existence of a public-private key pair for every principal, and assert that it is known in advance, along with the identities of all participants, their public keys, and any long-term keys the principal is party to.

The Evolution of Knowledge The `learned_times` relation indicates how an agent's knowledge grows over time. For every tuple (n, v, t), where n is a name, v is a term, and t is a timeslot, (n, v, t) is in `learned_times` if and only if n can derive v at time t from prior knowledge and any message received at t. Some caution is needed: naively, this can lead to self-justifying knowledge. We use an analogy to defining the transitive closure (TC) of a relation R in first-order logic. One might be tempted (especially after seeing the idea in Datalog) to write TC as:

$$\forall x, y \,|\, TC(x, y) \iff (R(x, y) \vee \exists z \,|\, TC(x, z) \wedge R(z, y))$$

Unfortunately, this sentence fails to encode that TC must be the *least* such relation. Similarly, suppose we state that (1) a ciphertext term may be known if an agent knows its contents and the appropriate public key; and (2) a term within a ciphertext may be known if an agent knows the ciphertext and the matching private key. Now it is consistent for any agent to know any value, provided they also know a ciphertext wrapping both the value and its own decryption key.

We might prevent this spurious knowledge by allowing only one such action per timeslot, but that solution would prevent fully learning from messages that contain decryption keys. Any agent receiving the two values k_1 and $\{k_2, \{k_3\}_{k_2}\}_{k_1}$ must be able to learn the innermost value k_3: the key k_1 can be used to decrypt the outermost ciphertext, which itself contains a new ciphertext and key k_2 to decrypt it, and so on.

Instead, we impose a *microtick* discipline, inspired by simulation tools like Ptolemy [43]. Microticks subdivide every timeslot, providing a frame that helps ensure that knowledge is well-founded. In any microtick, knowledge may be derived only if it was just received, was known in a previous timeslot, or has been decomposed from more complex terms in a *previous* microtick. Then (n, v, t) is in `learned_times` if and only if n has just received a message and v is in the current `workspace` for some microtick.

The challenge of bounds Since Kodkod, and thus Forge, uses a bounded relational logic (Section 3), there is an inherent incompleteness to its analysis. This includes not merely how many nonces may be generated, but also more subtle factors like the maximum term depth. Bounds also pose a user-facing challenge: at the moment, queries must provide bounds, which can require much effort and mental arithmetic to produce. Some of these issues could be ameliorated by taking advantage [39] of sorting information on terms, and others, such as the inherent bound on the number of timesteps, cannot.

5 Processing CPSA Declarations

Our prototype uses Racket macros to expand `defprotocol`s and `defskeleton`s to:

1. sort definitions (e.g., every `role` induces a new sub-sort of `strand`);
2. relation definitions (e.g., every role variable becomes a new relation that maps strands of that role to the variable's sort); and
3. relational formula sets (called *predicates* in Forge) that define the meaning of the protocol or skeleton.

The predicates for each protocol, skeleton, etc. can be invoked in queries, giving the user control over which aspects of the CPSA input to include in the analysis.

Because Forge builds atop Kodkod's formalism, it has only a notion of *relations*, atop which *functions* must be defined via constraints. This means that function application must be expressed via relational algebra—most commonly by using the join operator. For instance, constraints ensure that, if `s` is a member of the `strand` sort, then the expression `(join s agent)` evaluates to the agent running strand `s`. Likewise, in the Needham-Schroeder example, strands `s` of role `init` have a field `a`. This field is represented by a relation named `ns_init_a` and the value of variable `a` in `s` can be found via `(join s ns_init_a)`.

5.1 Deriving Relational Constraints

For every role `R` in protocol `P`, the translator produces a Forge formula (named `exec_P_R`) that constrains the behavior of every strand of that role. In the case of roles, the main bulk of the work lies in enforcing that all strands with that role obey the provided trace declaration. E.g., in the Needham-Schroeder initiator strand, the first event must send the term `(enc n1 a (pubk b))`, and so on. One subtlety here is that Forge's constraint language has no notion of a *term* in the sense of CPSA's algebra; it has only relations. The translator cannot speak of `(enc n1 a (pubk b))` directly to mean *the* result of encrypting `n1` and `a` with `b`'s public key. Consequently, we use existentially quantified variables to stand in for non-ground terms, and recursively traverse every event to ensure the proper ordering and nesting between events and terms.

For skeletons, the main challenge is in encoding the *maplets* that equate variables in the skeleton with terms over variables in strands that the skeleton contains. A responder point-of-view skeleton for Needham-Schroeder (Section 2.2) contains the variables `a` and `b` (names) and a text value `n2`. These are bound in the strand definition `(defstrand resp 3 (a a) (b b) (n2 n2))`, which says that the skeleton's `a` is the same as the responder strand's `a`, and so on. We enforce these via equality constraints on the field relations for the corresponding strand and skeleton variables.

Declarations within a skeleton, such as a unique-origination requirements, as well as listener-strand definitions, become quantified formulas as follows:

$$\texttt{(uniq-orig v)}\colon \exists! s : Strand \mid originates[s, v]$$
$$\texttt{(non-orig v)}\colon \forall s : Strand \mid \neg originates[s, v]$$
$$\texttt{(listener v)}\colon \exists t : Timeslot \mid (\texttt{Attacker}, v, t) \in \texttt{learned\_times}$$

That is, respectively, there is a unique strand that originates the value, no strand originates the value, and the value is compromised at some point. Should a `uniq-orig` declaration appears in a role R, rather than a skeleton, it applies locally to all strands r with that role:

$$\texttt{(uniq-orig v)}\colon (\exists! s : Strand \mid originates[s, v]) \wedge originates[r, v]$$

5.2 Queries and Predicates

Users write queries in terms of the base and derived specifications combined. Since every role and skeleton formula is a Racket value, queries can build on, deconstruct, or otherwise manipulate these formulas. A full query formula might then look something like the following parenthetical Racket expression:

```
(and wellformed                ; base constraints
     exec_reflect_init         ; initiator strand
     exec_reflect_resp         ; responder strand
     constrain_skeleton_ns_1)  ; responder point-of-view
```

where `wellformed` enforces the base specification, the two `exec_` predicates add strand roles, and `constrain_skeleton_ns_1` asserts that only protocol runs containing skeleton 1 and its declarations should be included.

Breaking the overall query into multiple predicates has several virtues. Not only does it ease debugging, experimentation, etc. but it is also how our prototype supports CPSA-style input files with many skeletons: queries reference the pertinent skeleton(s) and no others.

6 Visualizing Strands

Alloy comes with a directed-graph-based model visualizer that has not altered much over its lifetime. In Forge, we have instead integrated the Sterling visualizer [15]. While Sterling reproduces (a slightly more modern and attractive version of) Alloy's visualizer, it also enables scripting using JavaScript and React. Thus, anyone familiar with these widely-used systems can create custom visualizations for their domain. In spite of the theoretical literature on reasoning from diagrams [3, 47], we are largely unaware of other general model-finding tools that deliberately provide scaffolding for domain-specific visualization. (A notable exception is the GUPU pedagogic Prolog system [40], which lets users define visualizations over answer substitutions returned by the engine.)

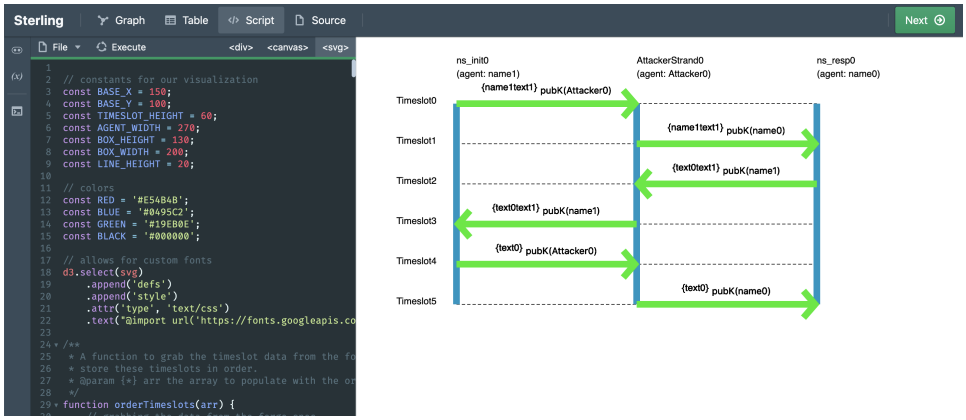


Fig. 5: The Sterling visualizer, with a Needham-Schroeder execution loaded.

Many protocol-analysis tools implement custom visualizations. Two of the most idiosyncratic are VerifPal [24] and CPSA [12], with VerifPal's more concrete display of (e.g.) agent knowledge contrasting against CPSA's minimal abstraction. We opted for a more concrete approach in order to showcase the power of custom visualizations. Figure 5 shows a full Sterling window containing a custom visualization. The left-hand pane shows the visualizer script being run—enabling changes without restarting either Sterling or Forge. The "Next" button advances to a different execution. Other key design choices include:

A Concrete Attacker Our visualizer shows the medium of communication explicitly as an attacker who, like others, can gain knowledge over time. One downside of this approach is that, depending on strand positioning, messages may be shown "crossing over" uninvolved strands. The attacker could be factored out in alternative visualizations (perhaps replaced with a terse "..."), but we found that an explicit attacker reinforces the Dolev-Yao adversary model.

Disambiguating Key Ownership Nonces, keys, agent names, and other data are represented in the visualizer by concrete atoms: `text0` might be a nonce or secret, `skey2` a symmetric key, `name1` the identity of an agent, and so on. Crucially, *atoms are not the same as* CPSA*-algebra terms*: while the terms `a` and `b` may denote the same value, the atoms `name0` and `name1` are necessarily different. This difference is especially important for key atoms. It would be baffling to see only that a ciphertext is encrypted with `akey3`—an asymmetric key, but whose? Because of this, our visualizer converts key atoms to equivalent CPSA-style terms whenever possible: e.g., `akey3` to `pubk(name0)` when `akey3` is `name0`'s public key.

Telescoping Knowledge State If a user is trying to understand *how* a certain attack occurs, information about agents' knowledge can be vital. Yet, showing *all* knowledge quickly becomes overwhelming. To mitigate this issue, we made the

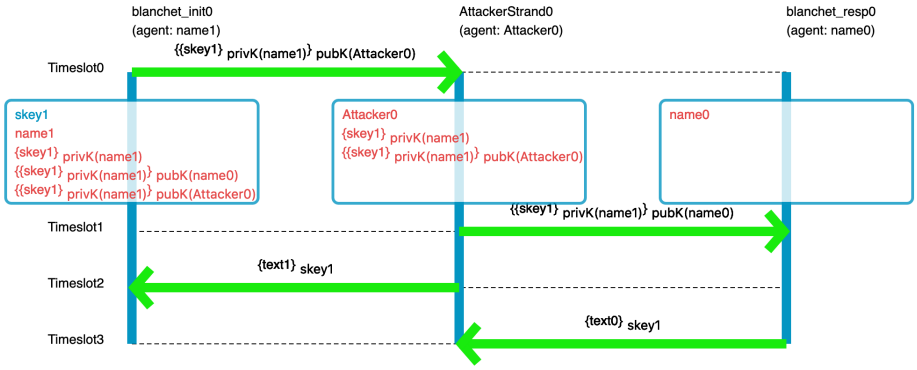


Fig. 6: A good execution of Blanchet's protocol with initial knowledge fully expanded. Freshly generated values are colored blue, and derived values (some unused) are colored red. Note also the nested ciphertexts, enabled by Section 4.

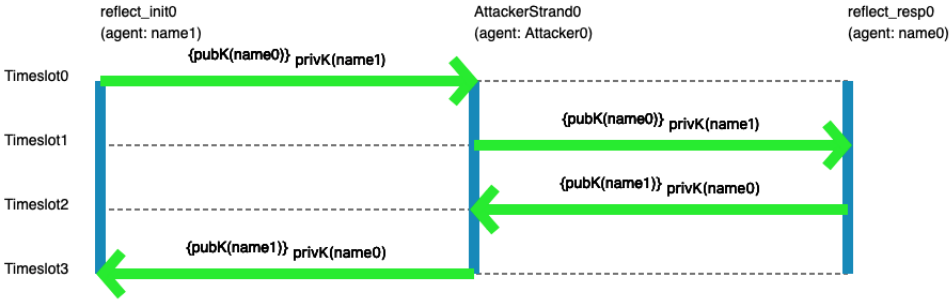


Fig. 7: A reflection attack from CPSA's example suite.

visualization *interactive*: users may click to expand (or hide) agent knowledge at any point in time. In Section 2, Figure 3 gave a magnified view of this feature; Figure 6 shows it in the context of a full visualization of Blanchet's [4] simple example protocol. While agent knowledge is not always pertinent to understanding an attack (as in the simple reflection shown in Figure 7), telescoping knowledge display makes it far easier to see how a value may be compromised.

7 Prototype Performance

Our goal is not to produce an optimized analysis tool, but rather a prototype that is "good enough" to iterate on. New language constructs, enrichments to the spec, or improved visualizations are all possible within the Forge framework. While some improvements such as custom search algorithms are not yet possible, many parts (especially the visualization) are portable. In addition, the prototype may be useful for validating a new, optimized engine via model-based testing.

Protocol	run	sat?	Runtime (sec)
Needham-Schroeder	validation	✓	5
Needham-Schroeder	attack	✓	6
Needham-Schroeder	non-attack	✓	5
Needham-Schroeder (fixed)	validation	✓	6
Needham-Schroeder (fixed)	attack		6
Reflection	validation	✓	4
Blanchet	validation	✓	4
Blanchet	init atk		5
Blanchet	resp atk	✓	4
Blanchet (revised)	validation	✓	4
Blanchet (revised)	resp atk		5

Fig. 8: Runtime performance (rounded to nearest second).

Despite these disclaimers, honesty compels us to report performance for all examples in this paper—extreme runtimes would undermine our stated goals. All examples were taken from CPSA's repository (Section 2) and run on a 2017 MacBook Pro (i5 2.3 GHz, 8 GB RAM). Concretely, we ran on:

- the original Needham-Schroeder [35] public-key protocol;
- Lowe's [26] modification to Needham-Schroeder;
- Blanchet's simple example protocol (from the CPSA manual [25]); and
- the "reflection" protocol demo (from the CPSA example suite).

These protocols exercise a number of core ideas in the basic CPSA algebra: asymmetric key pairs, short- and long-term symmetric keys, and nested ciphertexts.

For each protocol, we first ran a *validation* check to ensure the prototype found concrete executions. All validation checks were satisfiable. For Needham-Schroeder, we demonstrated the well-known attack and verified that the attack is no longer possible in the revised version. For Blanchet's simple-example protocol, we confirmed that the secret cannot be compromised from the initiator's perspective, but that it can be from the responder's perspective. We also confirmed that this vulnerability does not exist in the revised version.

Figure 8 reports results. The **Protocol** and **run** columns indicate which analysis was being performed. The **sat?** column denotes whether the analysis was satisfiable—i.e., whether any models were produced, or if the solver completed its search empty-handed. Finally, the **Time (sec)** column reports the runtime in seconds for the analysis.

Interpretation We find that runtime is largely uniform, in the single-digit seconds, across these simple protocols. This suggests that our (unoptimized) prototype scales reasonably to small examples. The CPSA analyzer is over an order of magnitude faster. However, roughly 2 seconds of Forge's time is spent on expanding the protocol and skeleton definitions and then compiling them to Racket bytecode. There may be strategies for reducing this overhead.

8 Related Work

Our end-to-end concept is partly inspired by Rosette [53], but differs significantly because of our focus on a direct encoding of domains in Forge, as well as our cultivation of domain-specific visualizations. Forge itself uses a heavily modified version of Rosette's Ocelot [6] interface to access the Kodkod [54] and Pardinus [10] relational solvers.

Our case-study prototype draws broadly from the strand-space formalism [51] and specifically from CPSA [12]. Strand spaces have been used to reason about a variety of protocols and related topics; a representative sample of which would include Guttman's work on fair exchange [20] and trust management [22]. Strand spaces also provide an interesting domain to ask foundational questions about model finding, chiefly *which* [14] models ought to be presented—a question that our prototype largely sidesteps in its present form.

There are of course several cryptographic analysis tools, such as VerifPal [24] and Proverif [4]. As our current effort focuses largely on strand spaces and CPSA's input language, from a specification perspective these other tools are largely unrelated. However, we note that Proverif's use of Horn clauses could potentially lend itself to similar prototyping in Forge. Even more, we drew inspiration from visualizations in other tools, especially Proverif, in building our prototype.

9 Discussion

We have presented the Forge system for prototyping solver-based DSLs atop Racket, and demonstrated its use in a prototype crypto-analysis tool in the vein of CPSA. The vast majority of the specification and visualization work was done by a pair of undergraduates over a (somewhat less than) one-semester course project. While we believe this work shows the viability of the approach, we would be remiss to close without first addressing a few limitations and sharing lessons learned beyond the trivial specification tricks seen in Section 4.

Fidelity w.r.t. CPSA We focused our effort here on sketching Forge's language-oriented prototyping process, rather than completely conforming to the semantics of CPSA. Further refinement along these lines (such as resolving limitations mentioned in Section 4, automated bounds inference, support for other algebras, etc.) would have been a matter of added engineering effort for little benefit: we have no desire to actually reproduce the already-excellent CPSA in Forge, but rather make an experiment in prototyping.

Which models? The question of *which* output model is beginning to be well studied: some works focus on minimality [37, 45], or closeness to a target [10, 27]. Other tools, like AUnit [49, 48] and CompoSAT [42] have prioritized models based on ideas from software-testing like *coverage* and *mutation*. Works like Bordeaux [34] have even argued for producing *non-models* to ease comprehension and debugging. We largely sidestep this question here, providing the user with

an Alloy-style "Next" button, but no further control. This can be frustrating, especially when compared against CPSA's sparse enumeration. We often found ourselves refining our queries and restarting the solver from scratch, rather than continuing manually. Thus, although we believe that CPSA's supreme abstractness can be a barrier to entry, especially for non-experts, we freely admit that its parsimonious output is more readily explorable at a high level than ours.

A Downside of Concreteness: Equality Since Forge produces models in terms of concrete atoms, it is free to have one atom serve multiple purposes unless prevented by the constraints it is given. Concretely, it might return first a run of Needham-Schroeder where the initiator and responder strands are hosted by the same agent, and then another run where the agents differ. This can lead to a plethora of seemingly spurious protocol runs, unless the user adds additional constraints to their query. In contrast, CPSA does not suffer from this issue: it will not equate two terms unless it can justify doing so. It would be informative to try this prototype using a different solver, perhaps one that is more amenable to an "enrich-by-need" analysis [14].

Forge and Solvers Forge currently uses only the Kodkod toolchain. Although it has recourse to weighted Max-SAT and other algorithmic extensions, it currently lacks a Satisfiability Modulo Theories engine. Forge is thus limited at present in its ability to reason about mathematical integers, strings, and other mainstays of SMT. Moreover, as we observed in Section 4, we needed non-trivial technical effort to even approximate CPSA's term algebras in Forge. However, we are encouraged by efforts to both translate relational specifications into SMT [19, 1, 30, 50] and encode a theory of relations directly in SMT [31]. As Forge's algorithmic capabilities evolve, so too will its capacity to be used as a prototyping framework; improvements to Forge would be immediately available to domain modelers and tool authors (Figure 1) via configuration options.

Acknowledgments We are grateful to Joshua Guttman for many enjoyable and productive conversations. We thank the creators of CPSA for their vision, the anonymous reviewers for their feedback, and the editors for putting together this much-deserved Festschrift. This work was partly supported by the US National Science Foundation. This research was also developed with funding from the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

1. Abbassi, A., Day, N.A., Rayside, D.: Astra version 1.0: Evaluating translations from Alloy to SMT-LIB. CoRR **abs/1906.05881** (2019), http://arxiv.org/abs/1906.05881

2. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The static driver verifier research platform. In: International Conference on Computer Aided Verification. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_11
3. Barwise, K.J., Allwein, G. (eds.): Logical Reasoning with Diagrams. Oxford University Press (1996)
4. Blanchet, B.: Modeling and verifying security protocols with the applied pi calculus and ProVerif. Foundations and Trends in Privacy and Security **1**(1–2), 1–135 (Oct 2016)
5. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Interactive Theorem Proving (2010)
6. Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Programming Language Design and Implementation (PLDI) (2017)
7. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous formal verification of Amazon s2n. In: International Conference on Computer Aided Verification (2018)
8. Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. In: International Conference on Computer Aided Verification (2018)
9. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: International Conference on Computer Aided Verification (2006). https://doi.org/10.1007/11817963_37
10. Cunha, A., Macedo, N., Guimarães, T.: Target oriented relational model finding. In: International Conference on Fundamental Approaches to Software Engineering. pp. 17–31. Springer (2014). https://doi.org/10.1007/978-3-642-54804-8_2
11. Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., Dougherty, D.J.: User studies of principled model finder output. In: Software Engineering and Formal Methods (2017)
12. Doghmi, S.F., Guttman, J.D., Thayer, F.J.: Searching for shapes in cryptographic protocols. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2007)
13. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Transactions on Information Theory **29**(2), 198–207 (1983). https://doi.org/10.1109/TIT.1983.1056650
14. Dougherty, D.J., Guttman, J.D., Ramsdell, J.D.: Security protocol analysis in context: Computing minimal executions using SMT and CPSA. In: International Conference on Integrated Formal Methods. Lecture Notes in Computer Science, vol. 11023, pp. 130–150. Springer (2018). https://doi.org/10.1007/978-3-319-98938-9_8
15. Dyer, T., Baugh, J.: Sterling: A web-based visualizer for relational modeling languages. In: Rigorous State Based Methods (2021)
16. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S.: A programmable programming language. In: Communications of the ACM (2018)
17. Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: A programming environment for Scheme. Journal of Functional Programming **12**(2), 159–182 (2002)
18. Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., Millstein, T.: A general approach to network configuration analysis. In: Networked Systems Design and Implementation. p. 469–483 (2015). https://doi.org/10.5555/2789770.2789803

19. Ghazi, A.A.E., Taghdiri, M.: Analyzing Alloy formulas using an SMT solver: A case study. CoRR **abs/1505.00672** (2015), http://arxiv.org/abs/1505.00672
20. Guttman, J.D.: Fair exchange in strand spaces. In: International Workshop on Security Issues in Concurrency. EPTCS, vol. 7, pp. 46–60 (2009). https://doi.org/10.4204/EPTCS.7.4
21. Guttman, J.D., Ramsdell, J.D., Wand, M.: VLISP: a verified implementation of Scheme. LISP Symb. Comput. **8**(1-2), 5–32 (1995)
22. Guttman, J.D., Thayer, F.J., Carlson, J.A., Herzog, J.C., Ramsdell, J.D., Sniffen, B.T.: Trust management in strand spaces: A rely-guarantee method. In: Schmidt, D.A. (ed.) European Symposium on Programming. Lecture Notes in Computer Science, vol. 2986, pp. 325–339. Springer (2004). https://doi.org/10.1007/978-3-540-24725-8_23
23. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, 2 edn. (2012). https://doi.org/10.5555/2141100
24. Kobeissi, N., Nicolas, G., Tiwari, M.: Verifpal: Cryptographic protocol analysis for the real world. In: International Conference on Cryptology in India. Lecture Notes in Computer Science, vol. 12578, pp. 151–202. Springer (2020). https://doi.org/10.1007/978-3-030-65277-7_8
25. Liskov, Moses D. and Ramsdell, John D. and Guttman, Joshua D. and Rowe, Paul D. : The cryptographic protocol shapes analyzer: A manual. https://github.com/mitre/cpsa/blob/master/doc/cpsamanual.pdf, accessed June 6, 2021
26. Lowe, G.: An attack on the Needham-Schroeder public-key authentication protocol. Inf. Process. Lett. **56**(3), 131–133 (Nov 1995). https://doi.org/10.1016/0020-0190(95)00144-2
27. Macedo, N., Cunha, A., Guimarães, T.: Exploring scenario exploration. In: International Conference on Fundamental Approaches to Software Engineering (2015)
28. Macedo, N., Guimarães, T., Cunha, A.: Model repair and transformation with Echo. In: Automated Software Engineering (2013). https://doi.org/10.1109/ASE.2013.6693135
29. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: Automated Software Engineering (2001). https://doi.org/10.1109/ASE.2001.989787
30. McCormick, K.D., Cinelli, F.C.: Translating Alloy to Smt-Lib. Major qualifying project (b.s. thesis), Worcester Polytechnic Institute (2018)
31. Meng, B., Reynolds, A., Tinelli, C., Barrett, C.W.: Relational constraint solving in SMT. In: International Conference on Automated Deduction (2017). https://doi.org/10.1007/978-3-319-63046-5_10
32. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A tool for generating structurally complex test inputs. In: International Conference on Software Engineering (2007)
33. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: A general-purpose higher-order relational constraint solver. In: International Conference on Software Engineering (2015)
34. Montaghami, V., Rayside, D.: Bordeaux: A tool for thinking outside the box. In: International Conference on Fundamental Approaches to Software Engineering. pp. 22–39 (2017)
35. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Communications of the ACM **21**(12), 993–999 (Dec 1978). https://doi.org/10.1145/359657.359659

36. Nelson, T., Ferguson, A.D., Scheer, M.J.G., Krishnamurthi, S.: Tierless programming and reasoning for software-defined networks. In: Networked Systems Design and Implementation (2014)
37. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: International Conference on Software Engineering (2013)
38. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: USENIX Large Installation System Administration Conference (2010)
39. Nelson, T., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Toward a more complete Alloy. In: Conference on Abstract State Machines, Alloy, B, and Z (2012)
40. Neumerkel, U., Kral, S.: Declarative program development in prolog with GUPU. In: International Workshop on Logic Programming Environments. pp. 77–86 (2002)
41. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. Communications of the ACM **58**(4), 66–73 (2015). https://doi.org/10.1145/2699417
42. Porncharoenwase, S., Nelson, T., Krishnamurthi, S.: CompoSAT: Specification-guided coverage for model finding. In: International Symposium on Formal Methods (FM) (2018)
43. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014), http://ptolemy.org/books/Systems
44. Rupakheti, C.R., Hou, D.: An abstraction-oriented, path-based approach for analyzing object equality in Java. In: Working Conference on Reverse Engineering (2010). https://doi.org/10.1109/WCRE.2010.30
45. Saghafi, S., Danas, N., Dougherty, D.J.: Exploring theories with a model-finding assistant. In: International Conference on Automated Deduction. pp. 434–449. Springer (2015)
46. Sergey Bronnikov: Practical FM. https://github.com/ligurio/practical-fm, accessed January 23rd, 2021
47. Shimojima, A.: On the Efficacy of Representation. Ph.D. thesis, The Department of Philosophy, Indiana University (1996)
48. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: Software Testing, Verification and Validation (ICST) (2017). https://doi.org/10.1109/ICST.2017.31
49. Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D.: Towards a test automation framework for Alloy. In: Symposium on Model Checking of Software (SPIN). pp. 113–116 (2014). https://doi.org/10.1145/2632362.2632369
50. Tariq, Khadija: Linking Alloy with SMT-based Finite Model Finding. Master's thesis, University of Waterloo (2021), http://hdl.handle.net/10012/16756
51. Thayer, F.J., Herzog, J.C., Guttman, J.D.: Strand spaces: Proving security protocols correct. J. Comput. Secur. **7**(1), 191–230 (1999)
52. Torlak, E., Bodik, R.: Growing solver-aided languages with Rosette. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. SPLASH Onward! (2013)
53. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: Programming Language Design and Implementation (PLDI) (2014)
54. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 632–647 (2007)