2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

# Advanced State Space Tunneling: Debug Your Formal Complexity Using Waveforms

Diego Hernandez[1], Erik Seligman[1], Lars Lundgren[1], Mariane Goncalves[1], Gustavo Junqueira[1], Tulio Leao[1], Gabriela Bahia[1], Håkan Hjort[1], Craig Deaton[1], Varun Ramesh[2]

[1] Cadence Design Systems, Various locations (Germany/United States/Sweden/Brazil); emails: {diego1, seligman, llars, mariane, gustavoj, tulio, gbahia, hhjort, cdeaton}@cadence.com

[2] Tenstorrent, United States; email: vramesh@tenstorrent.com

***Abstract*—One of the most challenging aspects of Formal Property Verification (FPV) is complexity: a subset of properties neither prove nor falsify but simply time out, with no waveform to debug. The cause is typically the same—missing context that would rule out unreachable states within a limited reasoning window. State-Space Tunneling (SST) turns that silent complexity into a minimal, readable trace. When the engine hesitates, SST requests the shortest legal trace in which the property stays good for k cycles and breaks at k + 1. The trace exposes missing assumptions or invariants that are missing. Once the engineer encodes this insight as a helper assertion and proves it, the proof often closes or the bound shrinks dramatically. This paper presents: (1) a practical walkthrough of SST; (2) recent tool features—soft-constraints, single command invocation, trace validation—that make SST usable in everyday flows; and (3) an industrial study at Tenstorrent where a two-hour nonconvergent run reduced to fifteen minutes after a single SST-guided helper, with similar multipliers on larger blocks.***

***Keywords*—Formal Property Verification, State Space Tunneling, invariants, proof convergence, proof complexity.***

## I. Introduction

One of the most challenging elements of FPV is when **complexity** strikes: some properties time out—neither proven nor disproven—and the tool offers no waveform to debug. Increasing compute resources rarely helps because the obstacle is fundamental: the formal model, extracted from the RTL, admits states that real hardware can never enter, and the engine lacks the context to rule them out.

There are many well-known techniques available to address FPV complexity, but they can be challenging to use because they lack the direct insight of a waveform. Bounded search which can only suggest "try deeper", counterexample waveforms appear only when the design is truly broken, and fully automated abstraction may hide the root cause. None of these methods explains **which missing fact** would allow the proof to finish.

**State-Space Tunneling (SST)**—SST fills this gap. It produces a minimal trace that: (i) begins in a window of **k** states where all assumptions hold, (ii) keeps the property good for those **k** cycles, and (iii) reaches bad at cycle **k + 1**.

The trace pinpoints where the sliding-window check first loses certainty. Reading it reveals the still-implicit rule. Encoding that rule as a helper assertion, proving it, and rerunning either closes the proof or advances the bound; iterating this loop typically converges quickly in practice.

**Paper outline**. Section II revisits the proof model and how SST fits. Section III shows how even a two-counter toy can hang an engine for millions of cycles. Section IV details the SST workflow and its effect on that example. Section V summarizes recent tool integrations. Section VI reports Tenstorrent results. Section VII presents limitations and scope. Section VIII concludes.

## II. Preliminaries

FPV asks a verification question: *Can any legal input sequence, starting from reset, reach a state that falsifies this property?* The tool answers with three outcomes: (i) exits with a proof, (ii) produce a counterexample waveform, or (iii) *stops as* **undetermined** *when resources expire*. The day-to-day bottleneck is the last case—proofs that stall silently. SST turns those stalls into actionable insight. The material below explains how the search works and where SST fits.

FPV is a powerful validation technique that can provide theoretically complete coverage, unlike the statistical coverage testing of simulation, which is why there has been rapid movement in the industry to applying it wherever practical. Before we get into the details of SST, it's useful to start with some basic foundations to clarify the discussion. If you are already familiar with FPV, you can skip down to §II-C.

### A. State Space

A synchronous RTL block's behaviour is captured by its **state vector**—the combined values of all flip-flops and latches at a clock edge. With $N$ storage bits, $2^N$ bit patterns are theoretically possible; this set is the **full state space**. The **reachable state space** is the much smaller subset the design can visit after reset under legal-input assumptions.

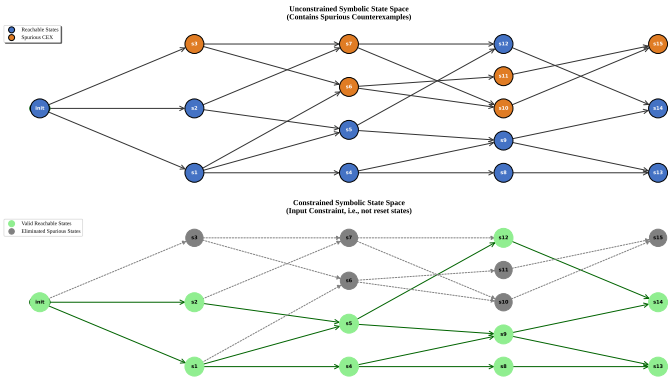


**Figure 1.** FPV engines typically constrain the initial state after reset, assigning specific values to state elements (blue/green), which prunes spurious bad states (orange/gray).

### B. Unbounded search

A property is a Boolean expression evaluated on a state. If true the state is **good**; if false it is **bad**. The goal is to show that *no bad state is reachable*. When achieved, the property is an **invariant** of the RTL (whether silicon implements the same model is handled later in the flow and is outside this paper scope). In practice, users encode such invariants as **helper assertions**, so they are checked and re-used automatically.

Modern FPV engines pursue the goal by repeating two bounded checks at an integer *bound* ***k***:

1. ***Reset window:*** unroll $k$ steps strictly from reset. All assumptions hold; check that the property stays *good* at every state**,** if this fails the engine returns a counterexample, otherwise continue.
    - ***Sliding-window:*** Forget the exact initial register values and allow any values consistent with assumptions. Require that the property holds for the last $k$ cycles as in (1), and ask: Can one more legal step break *good* states by transitioning to a *bad* one?
    - If **False**: Proof finished (states can only transition to *good*, the design can never step out of the *good* region, the design is safe forever).
    - If **True**: Raise $k$ and start another round.

If every state in every execution is good, the property is proved. The corresponding $k$ at which both steps succeed is the **upper bound**: (i) the reset window finds no counterexample and (ii) the sliding-window check is false. If resources expire before an upper bound is found or a real counter-example appears, the run ends undetermined.

### C. Where SST Fits

SST shows the scenario when the *sliding-window* check returns true or satisfiable as a concise diagnostic waveform: the first $k$ states satisfy the property, the *(k + 1)-th* violates it. This trace is not a design bug; it highlights an unrealistic scenario allowed by the property definition and RTL model that tells the user exactly which combination of states the engine cannot rule out. The engine needs to find an argument to prove why this scenario is impossible, normally by increasing $k$ and repeat the search. With SST, engineers inspect the waveform, add a

helper assertion, or tighten an input assumption, and rerun the proof loop—often at the same depth *k*, collaborating with the engine to find improvements that can lead to eradicate complexity and help convergence.

SST adds value to FPV by providing:

- **Visibility into stalled proofs** – SST replaces a silent status with a minimal, readable waveform.
- **Guided helper assertion development** – each SST shows exactly which unreal combination must be excluded.
- **Faster convergence** –helpers shrink state space. This often accelerates convergence for multiple related properties at once.

All subsequent sections build on this foundation: the reset window secures the chain of *good (k)-states*; the **sliding-window** check tests indefinite *self-preservation*, SST explains why the engine hesitates at current bound *k*; the loop iterates until either a real bug is found, or property converges.

## III. Illustrative Example

Figure 2 shows two 16-bit counters: cnt_a increments, cnt_b decrements. The *Property* requires cnt_a to increase and cnt_b to decrease when cnt_a is not all ones.

```
module example #(parameter W=16)
   (input wire clk, rstn, en);
   logic [W-1:0] cnt_a, cnt_b;
   always_ff @(posedge clk) begin
      if (!rstn) {cnt_a, cnt_b} <= {16'h0000, 16'hffff};
      else if(en){cnt_a, cnt_b} <= {(cnt_a+1'b1), (cnt_b-1'b1)};
   end
   Property: assert property (~&cnt_a |=> cnt_a > $past(cnt_a) -&& cnt_b < $past(cnt_b) );
endmodule // example
```

**Figure 2.** RTL for case study—16 bit counters and the *Property* assertion.

Though trivially correct, verification stalls. Figure 6 shows 2,441,956 cycles in 45 minutes without convergence. This exemplifies a common problem—the engine cannot identify necessary concluding argument, causing indefinite stalls even for simple designs.

Properties

| Type | Name | Engine | Bound | Traces | Time |
|---|---|---|---|---|---|
| Assert | example.Property | H | 2441956 - | 0 | 2714.6 |
| Cover (related) | example.Property:precondition1 | PRE | 1 | 1 | 0.0 |
| Assume | assume:0 | ? | | 0 | 0.0 |

**Figure 6.** The property panel shows how our target property has reached a bound of 2,441,956 in ~45 minutes without converging.

In next sections we show how State-Space Tunneling (SST) addresses this by improving proof convergence using a systematic method.

## IV. State-space Tunneling In Practice

In §II-B, the *reset-window* step confirmed "no bug up to cycle *k*" exists and *sliding-window* step attempted to show "if it is good for the last *k* cycles, it will stay good forever". FPV complexity appears when engine cannot rule out such states in an ever-increasing or stalled *k.* SST works as the translator that allows the formal engine to communicate with the human who's asking for certain proof—when the *sliding-window* cannot decide, SST asks the engine to speak back with a minimal, legal scenario that explains the hesitation.

SST does not return an arbitrary, unrealistic counterexample. It shows a case where cycles *1…k* satisfies every assumption and the property, but cycle *k+1* is the first legal step that breaks it. Therefore the waveform highlights the scenario that matters, exactly where the safety net tears, and the engine cannot easily rule out. Thus connecting the most fundamental part of unbounded analysis to an actionable, auditable debug tool.

A typical SST workflow looks as follows:

- **Trigger:** If the engine neither proves nor refutes the property after some bound k, invoke SST with that k.
- **Inspect:** Read the returned trace, identify the missing condition, and formulate a helper assertion.
- **Prove helper:** Prove the helper; once inductive it joins the invariant set.
- **Re-run:** Run the property again. If still undecided, iterate.

The exact same loop applies when a helper itself resists proof.

If SST finds no trace for the requested length, this means it accidentally discovered an upper bound—the property keeps itself *good* once the window is that long. Combine it with the lower bound to know the exact distance to closure. If SST presents a trace, the property is not self-preserving; the trace is the diagnostic log.

As a practical example, consider the design shown in Figure 3. SST returns the following waveform:

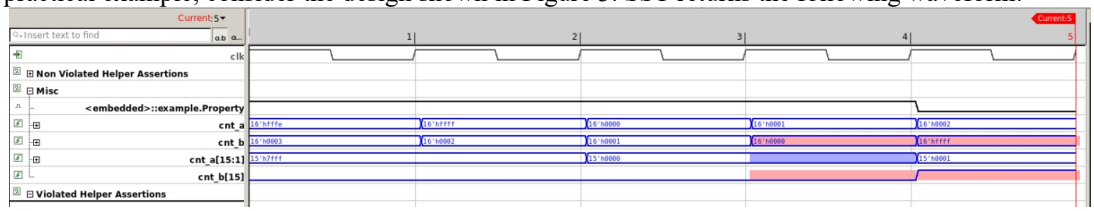


**Figure 7.** An SST trace that shows how *Property* can transition from *good* to *bad* in *k=5* steps. This trace works as diagnostic, it represents a case where the engine cannot conclude if the property holds with the current context.

Is the step at k+1 legal in real life?, If legal, is it acceptable behavior? (probably unconstrained design). If it is legal but undesirable, which condition should forbid it? (That condition is the helper assertion). Helper assertions shorten the symbolic highway the engine must explore. In the example of Figure 3, adding the helper assertion *cnt_a == ~cnt_b* collapses the search frontier; the next run closes in seconds, as shown in Figure 8.

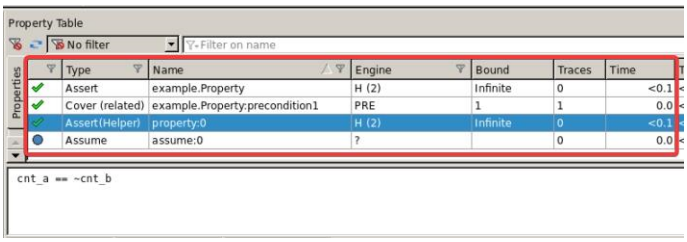


**Figure 8**. The helper lemma *cnt_a == ~cnt_b* is proven at *k=2* because, by the step 1 of Algorithm 1, the superset of all initial (but unconstrained) states under consideration already satisfies the helper at k=1. At k=2 the helper is again satisfied, and no new state is learned. The engine can see that, for *any possible* initial state where *Property* is good, all possible next states also preserve this status.

In summary, SST asks "I can get here legally; should I?" expecting the engineer to answer providing missing context (in the form of helper assertions). The helpers become an executable proof certificate: anyone rerunning proof sees exactly why the *proven* green tick is deserved. This dialogue style scales—from small flip-flop demos to SoC-size blocks, because the engineer improves the model, not the computational power required for proof.

## V. RECENT ADVANCES IN SST— *TOOL AND ENGINE-LEVEL INTEGRATION*

SST is part of the Jasper™ Technology portfolio. Recent SST updates have significantly improved usability. The following points describe some of those improvements:

**Integrated Prove Command (prove -sst):**

Previously, SST traces required separate visualization steps, limiting usability. Now, SST is fully integrated into the main prove flow, meaning it naturally fits standard verification iterations. Practical benefits observed include:

- Multi-property handling: simultaneous analysis simplifies verification of related properties.
- Background execution (prove -sst -bg): avoids blocking main workflow.
- Direct scripting and table integration: standard scripting commands (get_property_info, etc.) and the property-results table directly reflect SST results, improving clarity and consistency.

**Improved Debugging Capabilities:**

A particularly helpful improvement is the interactive modification of SST traces in Visualize™ (Modify Value, Replot). Engineers can quickly test alternative scenarios directly on the waveform, making complex debug significantly more intuitive and effective.

**Waveform clarity with soft constraints:**

Previously, SST waveforms from arbitrary states could become unintuitive and difficult to interpret. Now, SST uses soft constraints to gently guide traces closer to reset conditions. This produces waveforms more representative of realistic scenarios, making diagnostics faster, simpler, and clearer (see Figure 10).

**Trace Validation (sst -check):**

The new command to quickly verify if added helper assertions invalidate previous SST traces saves considerable manual effort. This ensures helper assertions consistently reflect design intent without unintended side-effects.

**Clearer Reporting in Property Tables:**

The explicit display of maximum achieved SST length when no violation trace is found gives a clear, practical indicator of progress towards full proof convergence (see Section 5.B. Defining SST) .

**Simplified Assertion Management:**

Now it's possible to convert regular assertions into SST-specific helpers (assert -set_sst, assert -clear_sst), facilitating quick refinement of helper assertions needed for efficient proofs.

**Clearer Proof Feedback:**

Enhanced messages during multi-property proofs precisely describe dependencies between properties. It is now easy to understand conditions under which certain properties have been conditionally proven, reducing confusion during complex analyses. For example, adding the helper *helper:0* to the example design without proving it, and asking for an SST trace for *Property* reports the lower bound and if any property is already proven during SST analysis:

```
Condition: none of the following properties have a trace shorter than 3 cycles:
    example.Property
Also: the following 1 properties are fully proven:
    property:0
```

**Figure 9.** Proof Feedback Messages.

These updates aim to reduce burden of adoption and enhance the practical debugging experience, making SST not just usable but genuinely efficient for everyday verification tasks.

**Figure 10.** Comparison between an SST trace without soft constraints (top) and an SST trace with soft-constraints near to reset, showing that using soft-constraints may render a more intuitive waveform.

## VI. INDUSTRIAL CASE STUDY – TENSTORRENT

At Tenstorrent, a production proof (with a FIFO involved) expected to converge fast, was taking around 2 hours to converge. The engine wasn't "slow"—it was exploring unreachable states because the property was too loose. One SST trace made the problem obvious—exposing an impossible "fifo output count > 4". Turning that single insight into a helper assertion instantly collapsed runtime from **>2 h to 15 min** (8×) .

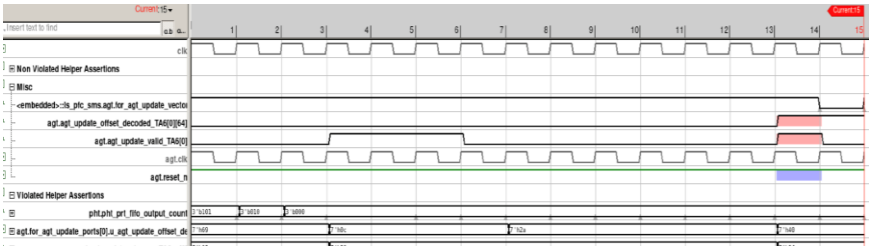


**Figure 11.** Example of an SST trace showing an unrealistic scenario. Once understood, the helper invariant improved the formal verification environment.

On a larger block, Tenstorrent team repeated the cycle: trace → ask "can this really happen?" → add helper → rerun: 20 helpers were discovered over a few weeks; each one fenced off another part of unreachable state. What had been a non-converging proof converged.

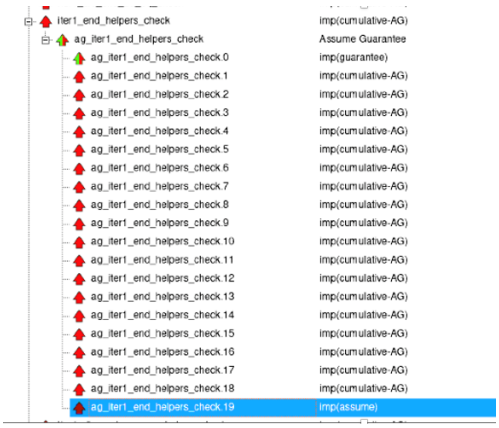


**Figure 12.** Iterative discovering of helpers in a proof-structure.

The point is not the specific helper (fifo_count <= 4). The value is in the **process**:

1. **Surface ambiguity**—SST starts from an arbitrary state and shows the first place where the assertion seeks nonsense behaviour.
2. **Trim the nonsense**—Encode "that can't happen" as a helper; prove it. Next search is more focused.
3. **Iterate** – Repeat until the main property is all that remains.

Instead of a black-box pass/fail, SST turns formal verification into a *conversation with the design.* Each trace asks, *"Did you really mean this?"* Each helper answers, *"No—keep the engine on the real highway."* The result is faster proofs *and* sharper specifications; engineers leave with stronger assertions and a clearer mental model of the design, not just a green tick.

## VII. LIMITATIONS AND SCOPE

SST is a visibility assistant around the k→k+1 check (§II-B, §IV); it does not guarantee that a suitable helper invariant exists in the user's template or that helper proofs will converge. Two failure modes are common: (i) **non-actionable traces**, where the minimal window points to, for example, combinations that require non-local/history or algebraic invariants; and (ii) **helper non-convergence**, where the proposed helper is as loose as the target property or each fix removes one trace but another appears, indicating the missing fact lies outside the chosen signals/states. In these cases, *change the expression of the helper* rather than increasing compute: switch

from local candidates to design-specific relations (bounds, pointer distance mod depth, history variables), add explicit environment assumptions when the trace exploits illegal handshakes, etc. SST still adds value by pinpointing *where* context is missing. The underlying limits are those of k-induction itself (e.g., deep history requirements and unreachable windows), as documented in the induction literature [2–6].

## VIII. CONCLUSIONS

Understanding proof closure improves the practical experience with formal methods and improves quality of results. SST transforms a silent stall into a waveform that asks a precise question: "All assumptions hold for k cycles; do you also forbid this next step?" Every answer—a helper assertion—tightens the model and brings the sliding-window test closer to a definitive "safe forever". In the tutorial example one helper collapsed a non-convergent 45-minute run to seconds; in the Tenstorrent case study a single helper invariant cut two hours to fifteen minutes, and a handful more converted an un-provable SoC block into a routine overnight check.

SST therefore serves two roles: debug microscope during development and proof certificate generator for tape-out sign-off. With recent one-command invocation and trace-validation features, the technique is now lightweight enough for daily use. The central lesson is already clear: make the engine talk, listen to the trace, turn it into a rule, and the chance of closure increases. Thus, it is now practical and efficient for everyday FPV users to leverage SST for waveform-based debug of formal complexity.

## REFERENCES

[1] IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Standard 1800-2017, IEEE Standards Association, Piscataway, NJ, USA, Feb. 22 2018.

[2] De Moura, L., Rueß, H., & Sorea, M. (2003, July). Bounded Model Checking and Induction: From Refutation to Verification: (Extended Abstract, Category A). In *International Conference on Computer Aided Verification* (pp. 14-26). Berlin, Heidelberg: Springer Berlin Heidelberg.

[3] Brain, M., Joshi, S., Kroening, D., & Schrammel, P. (2015, September). Safety verification and refutation by k-invariants and k-induction. In *International Static Analysis Symposium* (pp. 145-161). Berlin, Heidelberg: Springer Berlin Heidelberg.

[4] Vediramana Krishnan, H. G., Vizel, Y., Ganesh, V., & Gurfinkel, A. (2019). Interpolating strong induction. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31* (pp. 367-385). Springer International Publishing.

[5] Thalmaier, M., Nguyen, M. D., Wedler, M., Stoffel, D., Bormann, J., & Kunz, W. (2010, June). Analyzing k-step induction to compute invariants for SAT-based property checking. In *Proceedings of the 47th Design Automation Conference* (pp. 176-181).

[6] Roy, T., & Hsiao, M. (2017, October). Reachability analysis in RTL circuits using k-induction bounded model checking. In *2017 IEEE International High Level Design Validation and Test Workshop (HLDVT)* (pp. 9-16). IEEE.

[7] Chen, J., & He, F. (2020, December). Proving termination by k-induction. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (pp. 1239-1243).

[8] Biere, A., Heule, M., & van Maaren, H. (Eds.). (2009). *Handbook of satisfiability* (Vol. 185). IOS press.

[9] "JasperGold - State Space Tunneing (SST) Rapid Adoption Kit," Cadence, Tech. Rep., Aug. 2017.