Next-Generation Formal Property Verification: Lightweight Theorem Proving Integrated into Model Checking

Erik Seligman, *Cadence Design Systems,* seligman@cadence.com
Karthik Baddam, *Qualcomm,* kbaddam@qti.qualcomm.com
Barbara Leite Almeida, Thamara Andrade, Poliana Bueno, Carla Ferreira, Matheus Fonesca, Lars Lundgren, Raquel Lara dos Santos Pereira, Fabiano Peixoto, Vincent Reynolds, *Cadence Design Systems*

***Abstract*- In this paper we present a new FPV tool concept, "Proof Structure", which augments standard model checking to empower the user to carry out proof decomposition in a rigorous, well-defined way—with the tool given enough information to prove that the overall results are combined with correct reasoning, or to identify any holes in the process. This method effectively adds lightweight Theorem Proving to a model checking tool, though no user knowledge of Theorem Proving is required. We describe the main operations supported by our current implementation of Proof Structure and show how it has helped engineers at Qualcomm to achieve convergence on a difficult FPV problem, using multi-step decomposition and gaining high confidence in the overall soundness of their result.**

I. Introduction

Have you ever worked on a large formal property verification (FPV) problem, and had to decompose the problem into smaller pieces to make it possible? This is a natural technique used during modern design verification, and a necessity due to the inherent complexity of the "model checking" problem, the main problem addressed by current commercial FPV tools. Usually, such decomposition is handled by writing ad hoc user-level scripts, for cases where the underlying solvers have difficulty decomposing automatically. These scripts define the various pieces of the problem and assemble the results together to ensure the design is fully validated. However, this reliance on user-level scripts creates a key avenue for potential errors or escapes; indeed, we have seen numerous real-life cases where the subdivided problem was run perfectly on the tools, but an error was made during assembly or in porting specifications from a previous project. In this paper we present a new FPV tool concept, "Proof Structure", which augments standard model checking to empower the user to carry out this decomposition in a rigorous, well-defined way—with the tool given enough information to prove that the overall results are combined with correct reasoning or identify holes in the process. It can be described as a lightweight Theorem Proving layer added to a Model Checking tool; however, no familiarity with Theorem Proving is required for users. We describe the main operations supported by an initial implementation of Proof Structure and show how it has helped engineers at Qualcomm to achieve convergence on a difficult FPV problem, using multi-step decomposition, improving organization and performance, and gaining high confidence in the overall soundness of their result.

II. Motivation

As the capacity of FPV technology has grown, the capacity for directly proving requirements in FPV tools like Cadence Jasper has grown as well, but this has just led to users tackling bigger problems: decomposition has remained a critical part of proof methodology. These decompositions simplify the proof of a requirement, expressed as an *assertion* or *cover*. For example, one assertion could be divided into smaller assertions to prove in a Case Split, or helper assertions proven on parts of its input logic in an Assume Guarantee. Historically it has been taken for granted that the tool's responsibility is just to prove the specified assertions; assembling the pieces coherently into a solution of the original problem is beyond the tool domain. User scripts would create a "bag of tasks", essentially a bunch of defined subprocesses, each proving some subset of the assertions in a modified environment, that together make up the full proof. The user then had to ensure all tasks remain valid, logically relate to each other as envisioned, and are properly run, using their own expertise. This introduced a lot of risk at the user level, as gaps in the reasoning about how these tasks were constructed, and the relationships between them, were not checked formally by the tool. Examples of common mistakes in task-based reasoning include:

- **Ignored Properties.** A proof script might construct a set of tasks by explicitly listing the assertions to be proven in each task, and then reporting an overall success if all tasks pass. If a new RTL turnin (the submission of an updated set of design files to the repository) adds an embedded property not previously

assigned to any task, the script might continue to report overall success without ever trying to prove the new property.

- **Partial Assume Guarantee.** A common Assume Guarantee setup is to have one task prove the helper assertions, and another use those helpers as assumptions to prove the main target. Due to the challenges of complexity, FPV users typically allow some level of "bounded proofs", partial proofs that an assertion is true for <N> cycles after reset. These must be carefully reviewed before signoff. Suppose, for example, a helper only got a bounded proof to bound 50 in the helper task, but the target task reported a full proof assuming the helper. It would be very easy for a user to just look at the local task with the target proof, and incorrectly conclude that the target was fully proven. It can actually only be considered proven to the degree its assumption was proven—that bound of 50.
- **Incomplete Case Split.** When doing a case split, such as proving an assertion separately for each possible opcode, a user might not be aware that a new opcode was recently added and is not covered in their existing proofs. If the cases were listed directly in the proof script, the user might incorrectly continue to report that all cases are proven and the top-level property is fully true, without noticing the gap in their set of case proofs.

These types of issues are what inspired the new *Proof Structure* concept. Rather than just proving individual assertions, this new feature provides a rigorous, well-defined, and tool-visible way to define and carry out the decomposition strategies. (Some details and terminology here are rooted in the current implementation in the Cadence Jasper tool, but these principles could be used to enhance any model checking FPV tool.) The user decomposes the proof into a series of "operations", with each operation relating to the others and to the overall proof in well-defined ways. Any operation may in turn have sub-operations, definable using the same set of rules, in case the local proofs need to be further decomposed. Thus, the full set of decompositions can be thought of as a logical tree structure.

The tool maintains the decomposition tree and verifies that not only are the individual tasks executed successfully, but that they logically fit together in a way that coherently proves the original verification targets. Each property proof is analyzed in terms of the overall tree, and we only "propagate" logically valid proofs and counterexamples up the tree. Proof Structure will guard against cases where the user fails to run the decomposed jobs correctly, or a change in the design makes a current strategy invalid, notifying the user whether the pieces together still compose a full proof of the targeted assertions. The table below summarizes the change in approach:

Usage Element	**Traditional Model Checking**	**Model Checking with Proof Structure**
What to prove	Individual properties, as specified in user script commands.	Properties + composition rules showing proof can be propagated correctly.
Multiple proof tasks	User completely specifies environment for each task; tool has no knowledge of how tasks are related.	"Nodes" (Proof Structure tasks) are derived from other nodes using known decomposition rules, with tool-visible logical relationships.
Documentation of decomposition rules	Completely implicit in user scripts, no tool visibility.	Tree explicitly shows the decomposition from the base proof, inherently specifying how the pieces fit together.
Detecting logical gaps in decomposition method	Only possible with thorough manual review.	Proofs will fail to propagate up the tree if the nodes do not comprise a logically valid proof, automatically informing users of an issue.

III. A Simple Example

To illustrate the key concepts behind Proof Structure, let's look at a simple example. Suppose we have a 6-stage data pipeline design and want to formally prove that within an expected time interval of 6 cycles after we provide a valid input, a valid output will appear at the end of the pipeline. But our initial FPV attempt on the whole design results in an inconclusive FPV result, due to the logic complexity of the full pipeline. We then break it down with several complexity strategies:

- **Assume-Guarantee:** We want to provide "helper assertions" demonstrating that each half of the pipeline correctly transfers its data.

- **Underconstraint ("Stopat") #1**: The second half of the pipeline is complex on its own, so we want to cut out all logic from the first half, so the engines do not waste effort analyzing any of it during this proof. To do this we want to use a "stopat", telling the tool to cut off the driving logic at some point in the model (treating it as a primary input) to simplify the analysis.
- **Underconstraint ("Stopat") #2:** When proving the top-level pipeline correctness by assuming the two helper properties, we want to exclude the internal logic of the pipeline halves.

This strategy is illustrated below.

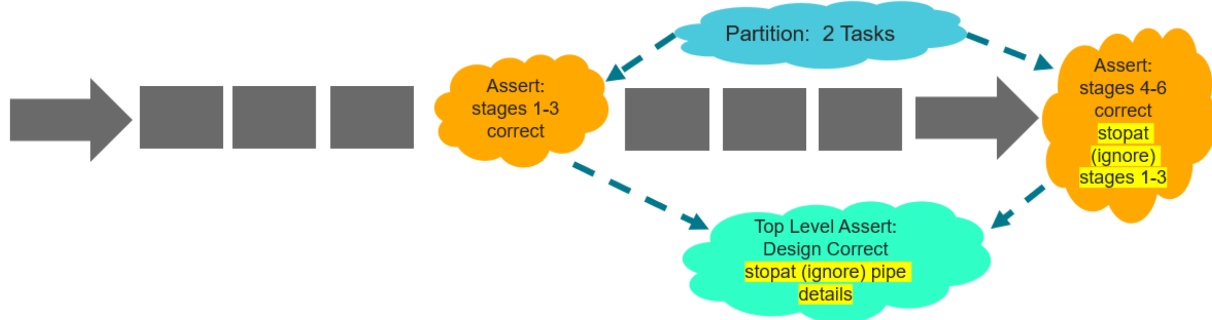

Figure 1: Simple proof where Proof Structure is useful.

Even though this is not a very complex decomposition, it typically requires the creation of at least three "tasks" in an FPV tool, separate proof environments, since each strategy above changes the set of assumptions and logic being examined. We need to partition the two helper proofs into separate tasks and create a top-level task that changes the helpers from assertions to assumptions to enable to final proof. Without Proof Structure, the FPV tool command file will end up looking something like this (specifying in tool-neutral pseudocode):

```
property target: (in_valid) |=> ##6 (out_data == $past(in_data, 6)
property help1:  (in_valid) |=> ##3 (pipe_data[3] == $past (in_data,3)
property help2:  (pipe_valid[3]) |=> ##3 (out_data == $past (pipe_data[3],3)

create_task helpers -from top -copy help1 help2
    assert help1
    assert help2

create_task help_task1 -from helpers -copy help1
    prove help1

create_task help_task2 -from helpers -copy help2
    stopat misc_early_pipe_logic
    prove help2

create_task top_assume_guarantee -from top -copy help1 help2 target
    assume help1
    assume help2
    assert target
    stopat misc_pipe_logic
    prove target
```

This use of multiple tasks can be very error-prone, as the idea that the tasks fit together to form a sound proof methodology is implicit, and not formally proven, in the script the user provides to the tool. As discussed in the risks listed in the previous section, the owner could look at the top-level task result, report that the overall pipeline is fully proven, and not notice that the helper task only reports a limited proof bound. Or a later user could misunderstand the script, "speed things up" by only running the *top_assume_guarantee* task, which proves the top level target assertion, and sign off on validation-- without realizing that this proof is only valid if the helpers have also been proven in the other tasks. And of course, in real-life examples of similar proof decompositions on large and critical design blocks, the breakdown may be into dozens of tasks, magnifying these dangers exponentially.

Let's see how Proof Structure solves these problems. Below we see the proof tree created with Proof Structure, where the proof is broken down into a series of "nodes".

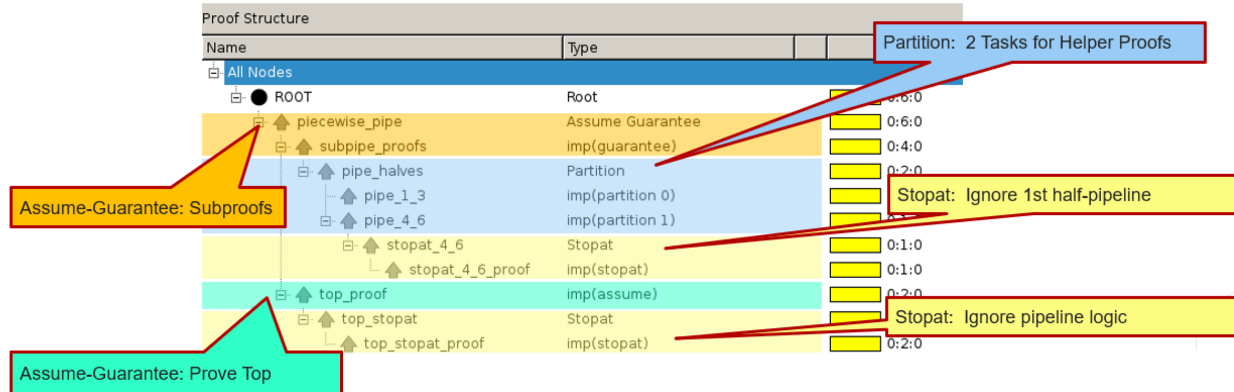

Figure 2: Proof Structure tree for Figure 1.

As you can see, each of the steps of the complexity decomposition we mentioned above is represented in the tree structure. From the *ROOT* node, which represents the original proof environment, we derive the *piecewise_pipe* Assume Guarantee "operation node". That operation contains two "implementation nodes", *subpipe_proofs*, which proves the helper assertions, and *top_proof*, which uses the helpers to prove the main target. Under *subpipe_proofs,* we have a Partition operation to separate the helpers into two subnodes, *pipe_1_3* and *pipe_4_6*. In *pipe_4_6,* we add the Stopat underconstraint, to generate node *stopat_4_6_proof.* Separately, at the top level, we add a different Stopat to generate *top_stopat_proof.* The above script would become something like this (again, in tool-neutral pseudocode here):

```
property target: (in_valid) |=> ##6 (out_data == $past(in_data, 6)
property help1:  (in_valid) |=> ##3 (pipe_data[3] == $past (in_data,3)
property help2:  (pipe_valid[3]) |=> ##3 (out_data == $past (pipe_data[3],3)

assert target
assert help1
assert help2

proof_structure_create_root ROOT
proof_structure_op piecewise_pipe -from ROOT -nodes {subpipe_proofs top_proof}
       assume_guarantee {help1 help2} {target}
proof_structure_op pipe_halves -from subpipe_proofs -nodes {pipe_1_3 pipe_4_6}
       partition {help1} {help2}
proof_structure_op stopat4_6 -from pipe_4_6 -nodes {stopat_4_6_proof}
       stopat misc_early_pipe_logic
proof_structure_op top_stopat -from top_proof -nodes {top_stopat_proof}
       stopat misc_pipe_logic
```

While this might look slightly more complicated at first glance, the key point is that the full logical structure of the proof decomposition is now visible and enforceable by the tool. We have not just created a bunch of tasks: the relationships between the tasks, such as the fact that *top_proof* is in an Assume Guarantee where its proofs are conditional on those in *subpipe_proofs*, are fully defined. This breakdown offers several advantages for ensuring proof soundness:

- **Consistent Environments**. When deriving a subnode from a node, proof environment changes (new assumptions, underconstraints, etc.) are fully known to proof structure: attempting any tool commands that would change the environment within a proof structure node is disallowed by the tool. This contrasts with the linear script style in the non-proof-structure version, where literally anything can be changed from line to line during the series of proofs, modifying assumptions, clocks, proof targets, etc. Thus, we ensure no user mistakes can unintentionally create inconsistent environments between nodes.

- **Known Propagation Rules:** In order for an assertion proof result in a subnode to "propagate", or be copied, to the parent node, it must follow soundness rules appropriate to the node type. For an Assume Guarantee operation node like *piecewise_pipe* above, it will only show the top level assertion as proven if both the helper assertions and the target assertion have passed in their appropriate subnodes. If one of the helpers are not proven, for example, it will still show the main target result as unknown at this level. Ultimately, if a property shows up as proven at the ROOT level, this means that the tree subnodes together constitute a valid proof.
- **Correct Bounded Propagation:** When bounded proofs are used, Proof Structure's propagation rules intelligently comprehend the bounds in each subnode and propagate a correct bound up the tree. For example, suppose the Assume Guarantee gets a full proof of the target assertion in the *top_proof* Assume subnode, but one of the helpers was only proven to bound 100 in the *subpipe_proofs* Guarantee subnode. Proof Structure correctly shows in the *ROOT* node that the propagated proof of the target is effectively only a bounded proof, to bound 100, since its full proof depended on the bounded-proven helper. Without Proof Structure, this subtlety is often missed in handwritten proof scripts, since the standalone task reporting the target result locally shows a full proof.
- **Correct By Construction**: Without Proof Structure, it is very cumbersome for users to manage their numerous tasks and keep track of how they fit together to decompose the original proof. Because the node types provided by Proof Structure are created with well-defined decomposition rules and sound propagation rules, and arbitrary manual changes to the proof environments are disallowed, we avoid dangers of users creating a set of tasks that do not together build a logically consistent proof. This also enhances user confidence in defining complex multi-task proof strategies, which many have previously been reluctant to do due to the high potential for human error. Indeed, some users have directly commented that once they could access Proof Structure, they used some complex decomposition strategies, with multiple Assume Guarantee layers, that they previously avoided simply due to the potential complexity and risk of their handwritten scripts.
- **Enable Parallelism**: Without Proof Structure, it has been common for user scripts to unnecessarily serialize execution of a series of proof steps, both for minimizing script complexity and out of fear of accidentally skipping logical requirements in the overall reasoning. However, once a proof tree has been defined in Proof Structure, you can run proofs on all the leaf node proofs in parallel, using grid/cloud features of the tool. If, for example, an Assume Guarantee target is proven before its helpers, that is fine: the propagation rules will ultimately only report a fully proven result after all logically required steps eventually complete. For this reason, use of Proof Structure can dramatically improve overall proof performance in practice.

III. Operations

As we have seen in the example above, the Proof Structure concept is built around specifying a tree of 'operations', known proof decompositions with well-understood logical rules, rather than the current industry standard of specifying a linear set of script commands. A nearly unlimited number of operation types is theoretically possible; when developing an implementation of Proof Structure in a tool, the most important challenge is to choose a solid set of operations that encompass widely used decompositions. The main Proof Structure operations we have currently selected to implement include:

- **Partition**. Divide the assertion set into groups, targeting each group in a separate subnode. Such partitioning is often helpful if different subsets of the assertions in a module require different proof strategies. This operation seems logically trivial, but can help prevent a common mistake: if new RTL or validation code is turned in that adds an assertion which was not already accounted for in the script (and thus placed in one of the defined partitions), the tool can warn the user, unlike in a typical task-based script, which would quietly ignore such cases. Alternatively, an option can be used to automatically place all unlisted assertions into a separate partition.
- **Underconstrain/Stopat.** Cut or generalize the logic to reduce analysis for the tool. This includes stopats, abstractions, or removal of assumptions. Proofs are fully valid under such underconstraints, though counterexamples may be questionable.
- **Overconstrain.** Hardcode some inputs or register values for certain proofs, or otherwise limit the behaviors of the model more than the initial environment would imply. This includes removing stopats or abstractions, adding assumptions, or tying a signal to a constant. While proofs are not valid in an

overconstrained environment, this can be very useful in project stages where formal bug hunting is the emphasis since counterexamples are valid.

- **Assume Guarantee.** Prove helper assertions on driving logic, then use them as assumptions for downstream proofs. This can be a single guarantee-assume node pair, or a chain of such nodes in which each one provides helpers for the next.
- **Case Split.** Run subproofs for different values of a key variable or expression, putting them together as a full proof. We provide both Soft and Hard variants of Case Split:
  - **Soft Case Split**: Prove the target property using each case as a precondition. So, if the cases are C1, C2 and the target property is T, subnodes prove *(C1|->T)* and *(C2 |-> T)*.
  - **Hard Case Split:** Prove the target property while assuming each case. So, in the same example, we would generate two subnodes containing (*assume C1; assert T)* and *(assume C2; assert T).*

  With Case Split, Proof Structure also generates some additional needed proofs. Both Soft and Hard Case Split generate a "completeness node", proving that at any given time, at least one of the cases needs to be true. For Hard Case Split, we also provide a "validity node", to prove that despite the overconstraining of the individual cases, the full set of case nodes does indeed prove the target.
- **Compositional Assume Guarantee.** Use a set of assertions as helper assumptions for each other. At first this might sound like circular reasoning, but it is not, because they are using each other inductively:
  - **Base Case**: First prove that all the assertions listed are independently true at bound 1, just out of reset.
  - **Inductive Step:** Assuming all the assertions are true up to bound N, prove each one is true at bound N+1.

As discussed above, each of these methods has associated hazards which, when they are implemented manually, can result in a logically incorrect proof overall, or untrustworthy results for some properties. These mistakes are especially prevalent in script-based methodologies when an existing proof is ported to a new project, and the new owner doesn't fully understand the original method. Even within a project, a validator writing task-based scripts may miss a subtle aspect of the proof strategy in a script developed months earlier, and accidentally make an update that breaks the proof. Due to its correct-by-construction nature and logically valid propagation rules, Proof Structure prevents these mistakes. If an assertion shows a propagated result at the root of the Proof Structure tree, the nodes below have consistently generated that result through logically valid inference rules. The table below summarizes the main propagation rules for the operations above:

Operation	**Proof Propagation Rules**	**Bound Propagation Rules**
Partition	No environment change- propagate everything	No environment change- propagate everything
Underconstrain	Propagate proof, don't propagate cex	Propagate any bound directly
Overconstrain	Propagate cex, don't propagate proof	Don't propagate
Assume Guarantee	Propagate any cex, only propagate proof if helpers proven	Propagate min bound of (target, helpers) as target bound
Soft Case Split	Propagate any cex. Propagate proof if all cases + completeness proven.	Propagate min bound of (min case bound, completeness)
Hard Case Split	Propagate any cex. Propagate proof if all cases + completeness + validity proven.	Propagate min bound of (min case bound, completeness, validity).
Compositional Assume Guarantee	Propagate cex. Propagate proof if all assertions proven.	Propagate min bound of all assertions.

For the sake of brevity, we have concentrated on assertion proofs in the above table, though similar rules apply to cover property traces. Wherever we can propagate an assertion proof, we can propagate an unreachable cover proof; and wherever we can propagate an assertion counterexample, we can propagate a reachable cover trace.

Note that there are also many enhancements possible to the rules above. For example, in the future we plan on implementing *bidirectional propagation,* so a proof can propagate up to a parent node and then down to a sibling with a similar environment—this may optimize performance in cases where multiple subnodes of an operation share some related properties.

By the way, observing the list of operations, we can see that one other advantage of Proof Structure is that it provides a straightforward way to set up some proof methods that are otherwise difficult to script. In particular, Soft Case Split and Compositional Assume Guarantee are both cumbersome to script on top of a basic FPV tool, but easy to use when built into a proof as part of a Proof Structure tree.

IV. RESULTS

The Proof Structure feature is now being used by numerous companies. At Qualcomm, it was recently leveraged for proving correctness of a Table Walk design. The chained Address Translation design style, as shown below, naturally inspired a chained Assume Guarantee decomposition for the proof.

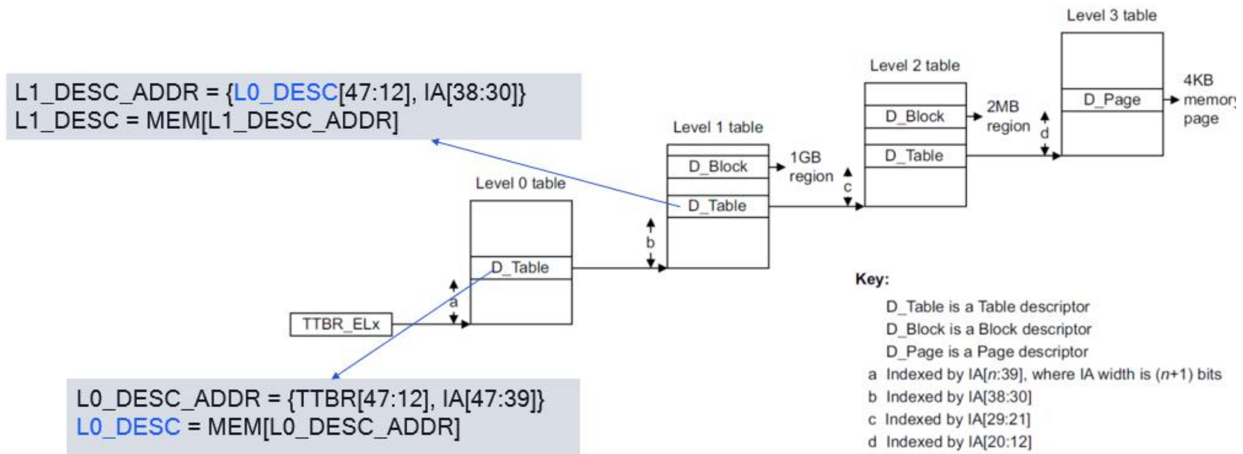

Figure 3: Qualcomm's address translation proof challenge

Initially this work was attempted using scripting, but the validator was nervous about possibly using helpers that had not been proven, so began an effort-intensive task of writing the script with careful sequencing of proofs rather than creating dozens of tasks to keep track of in parallel. This resulted in a complex script that was very cumbersome and far from optimal in performance. Switching to Proof Structure, he was able to cleanly define this proof using a multilayered Proof Structure tree. Naturally, with 50+ subproofs, a full walkthrough of this tree is not feasible in a paper of this size. But to give you a flavor of the decomposition they used, Figure 4 shows a view of part of their tree in the Jasper gui, along with the expansion of one of its subnodes.

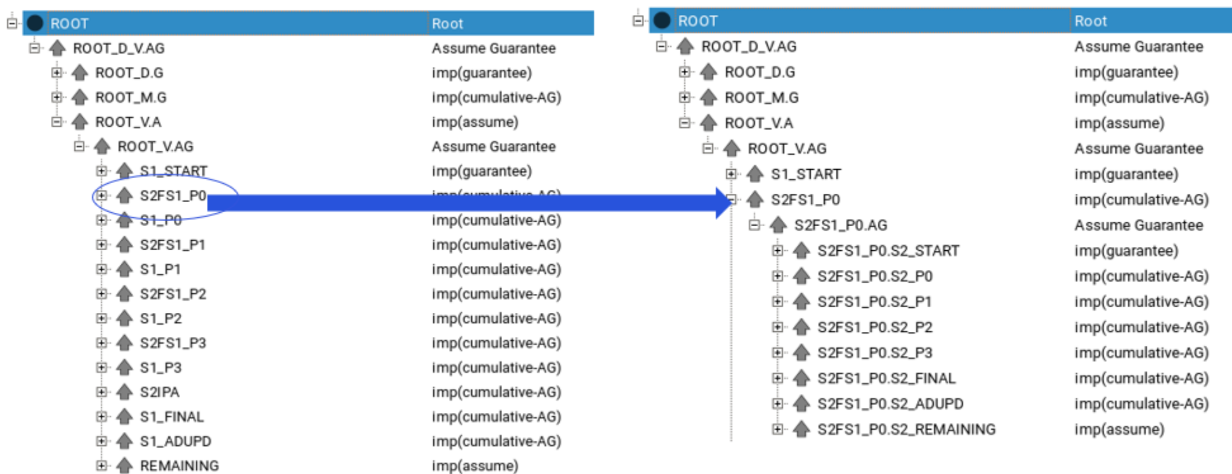

Figure 4: Part of the Proof Structure created by Qualcomm

As you can see, this involved multiple layers of Assume Guarantee: the helpers in *SF2S1_P0*, for example, were required to prove the ones in *S1_P0*, but themselves needed to be broken down into another Assume Guarantee

chain under *SF2S1_P0_AG*. In addition to the benefits of this cleanly structured decomposition of the Assume Guarantee chains, when transferring the environment to Proof Structure, the validator discovered several cases where the leaf level nodes were appropriate targets for Compositional Assume Guarantee, due to having some complex assertions with interdependent logic. Adding this operation enabled much faster proofs on these subnodes.

This Proof Structure version of the proof then provided a significant productivity boost. The final Proof Structure performed much better than the original script: the run had previously taken multiple days for each iteration, but now could complete in less than a day. This was a combined benefit of the improved parallelism and the use of Compositional Assume Guarantee in appropriate leaf nodes. In the end, this environment discovered 7 logical bugs in the design, including one which had actually been present for two years, but missed by all simulation and emulation environments.

In addition to this success, there have been several other reports of the benefits of Proof Structure in other recent conferences. These include:

- In [2], another team at Qualcomm describes a proof effort that leveraged Proof Structure. Like the above example, they found that the ability to define a proof decomposition tree and run all leaves in parallel was a major performance improvement over their previous serialized script, which they had considered too risky to parallelize by hand. They found six high-quality bugs using the method.
- In [3], Marvell reports on using Proof Structure Assume Guarantee and Case Split to set up a large architectural proof environment with over 10,000 properties. They estimate that Proof Structure saved them weeks of effort—and caught some logically incomplete cases in their script setup, which they would have likely missed without Proof Structure's automated case completeness checks.
- In [4], HPE reports an example where they used a Proof Structure with multiple layers of Assume Guarantee, Partition, and Stopat on a pipelined register access design. Proof Structure helped significantly with the logistics of setting up the proof environment, and led to the discovery of 15 bugs, including two long-existing bugs that had been consistently missed in simulation and emulation.

V. CONCLUSIONS

We have introduced Proof Structure, a new concept by which FPV tools can not only prove individual assertions but can also prove that sound decomposition methodologies were used to construct and combine the assertion proofs. Effectively we have provided a schema for integrating lightweight Theorem Proving into standard model checking tools, without requiring any user knowledge of Theorem Proving. The benefits of this method include:

- **Ease of Constructing Proofs**. Rather than being dependent completely on user-level scripts, the tool helps organize the overall strategy of the proof and maintains it as a visible tree that can be examined in the gui. This can noticeably increase productivity when implementing a complex decomposition method, as well as enabling users to run the subproofs in parallel with confidence. In addition, for some techniques like Soft Case Split and Compositional Assume Guarantee, it can be much easier to use the Proof Structure features than to script by hand.
- **Proven Soundness**. For a complex decomposed proof with many logical fragments that together comprise the overall argument, it is a critical benefit that the FPV tool itself can now prove the soundness of the proof strategy. When constructing large multi-task proof strategies in a handwritten script without Proof Structure, there are many ways to accidentally compromise overall soundness that can be difficult to detect. Proof Structure alerts users to flaws in their compositional reasoning, such as omitted Case Split cases or incomplete Assume Guarantee helper proofs, that they might have accidentally missed, and uses well-defined propagation rules to ultimately report logically valid results.
- **Maintainability and Reuse.** Verification engineers can now have confidence that subtle changes to the design or properties have not invalidated a previously sound proof strategy. If there are issues, the tool will report them automatically. They can also modify the proof strategy as the design or requirements change, with much less risk than they would have previously faced when editing a lengthy command script in late stages of a project. This significantly improves the ability to evolve and reuse FPV environments or hand off proofs to new owners with confidence.

This feature is now in use throughout the industry. With the example of its success at Qualcomm, as well as other successes reported by other users at recent conferences, we have shown that the new Proof Structure concept can easily be integrated and relied upon as a vital tool to increase the overall productivity of FPV.

REFERENCES

[1] Karthik Baddam, “Architectural Verification of MMU Address Translation in ARM CPU”, Cadence Connect: Jasper User Group 2023, October 2023.
[2] Nitish Sharma and Venkata Nishanth Narisetty, “Assume-Guarantee Proof Schema for Liveness Full Proofs”, Design and Verification Conference (DVCon) 2024, March 2024.
[3] Shahid Ikram and Erik Seligman, “A Leap Forward in Architectural Formal Convergence Using Automated Case Splitting”, Design Automation Conference (DAC) 2023, July 2023.
[4] Fernando Vargas and Jim Kasak, “Don’t Panic If You Have a Bounded Proof: Using Proof Structure with Assume-Guarantee to help with convergence”, Cadence Connect: Jasper User Group 2022, October 2022.