The Burden of Proof: Automated Tooling for Rapid Iteration on Large Mechanised Proofs

Chengsong Tan*†, Alastair F. Donaldson*, Jonathan Julián Huerta y Munive‡, and John Wickerson*
*Imperial College London, UK. Email: {c.tan, alastair.donaldson, j.wickerson}@imperial.ac.uk
†Kaihong, China. Email: tanchengsong@kaihong.com
‡Czech Technical University, Czech Republic. Email: huertjon@cvut.cz

***Abstract*—We report on challenges and solutions in making large mechanised proofs scale, based on our experience proving correctness properties for a cache coherence protocol. This was a difficult proof that required dozens of iterations to get right, and ultimately led to an inductive invariant with nearly 800 conjuncts, and to over 54,000 proof obligations. To address these proof engineering challenges we developed `super_sketch`, a tool that automates the generation of proofs involving multiple subgoals in Isabelle/HOL, enabling efficient management and maintenance of large-scale proofs. We further contribute `super_fix`, a tool to fix corner cases that cannot be fully automated with `super_sketch`, such as correcting proof scripts invalidated by upstream changes to definitions. This allowed us to drop simplifying restrictions in our model while retaining the correctness proof, as we avoid the significant manual effort that would otherwise be required to inspect and fix the broken lines of the original proof when generalising the model. Our work provides insights into proof engineering practices and highlights the need for improved support in proof assistants for large-scale mechanized proofs.**

***Index Terms*—Proof engineering, proof automation, mechanised proof, Isabelle, cache coherence, CXL, SWMR.**

I. Introduction

There is a growing need for better automation in interactive theorem provers (ITPs) [1], [2], [3], [4], [5], to enable formal verification at greater scales. Large mechanised proofs can be up to hundreds of thousands of lines of code, often taking many person-years, or even person-decades, to develop [6], [7], [8]. Although most of the proof engineering is mentally engaging, a considerable amount of time is spent on tedious tasks such as confirming that (often trivial) individual subgoals can be proven after applying a proof method like induction or case analysis, or fixing broken proof scripts that fail due to superficial changes in the formalisation.

Isabelle is a popular interactive theorem prover thanks to its powerful automation tools. For example, the Isabelle command `sledgehammer` invokes solvers to generate proofs for the user's theorems automatically. Despite `sledgehammer`'s usefulness, the user still needs to wait for a long time (often tens of seconds, sometimes minutes) for the utility to compute proof suggestions. The user must then manually choose one of the supplied proofs to adopt into their proof script. This process is often repeated multiple times because a theorem usually consists of various subgoals and `sledgehammer` only works on one at a time. This can be frustrating for a human expert who has already devised a correct high-level argument to prove a theorem: they nevertheless need to invest time and effort harnessing `sledgehammer` to fill out the easier (yet tedious to formalise) details of the proof. It would be beneficial if the generation of these parts of the formalisation could be fully automated by: (1) calling `sledgehammer` for each step in a formal proof-sketch, (2) extracting from `sledgehammer` the proofs it found, and (3) incorporating them directly into the sketch, while (4) highlighting unproved steps so that users can conveniently focus on them.

We faced this problem while doing a large mechanised proof of properties of a cache coherence protocol [9], which required us to generate the proofs of a large number of lemmas. Towards the end of the proof, we needed to prove over fifty thousand subgoals. However, we did not just need to mechanically check them once, but dozens of times, as we continuously refined our argument towards proving our desired theorem. These multiple cycles of proof attempts were needed due to our theorem hinging on a large inductive invariant comprising many conjuncts. It took us a long time to get the inductive invariant right. We would repeatedly find that the invariant was *not quite* preserved by the transition relation of our cache coherence protocol model: some conjuncts of the invariant would fail to hold after applying the transition relation. This would necessitate strengthening the invariant via additional conjuncts, but these additional conjuncts would then turn out not to be preserved by the stronger invariant, necessitating further strengthening. In the process of driving our proof towards convergence the number of proof goals ballooned, leading to each iteration of the process taking a large amount of human effort and machine time.

Performing a mechanised proof at this scale using standard tools became infeasible. We spent extensive machine resources (and wall clock time) waiting for `sledgehammer` to reprove subgoals from earlier iterations. We also expended a great deal of manual effort identifying broken lines in a proof and then calling `sledgehammer` to obtain a fresh sub-proof of the affected subgoal to rectify this problem.

In response to this, we have developed two tools to largely automate this process, allowing proof engineers to rapidly iterate on large mechanised proofs. These tools proved indispensable in enabling us to finally prove the desired cache coherence property of interest, which involved an inductive invariant comprising nearly 800 conjuncts, requiring over 54,000 proof goals.

We expect that our proof-effort details and scalability tools will be interesting and useful to others who embark on large mechanised proofs, hence this experience report.

Contribution 1: A report on the challenges of developing a large-scale mechanised proof of emerging hardware's correctness. Over the last two years, we have formalised a proof in Isabelle/HOL consisting of 74 Isabelle theory files, totalling around 310k lines of code. It certifies that our CXL (Compute Express Link [10]) model, an important industry interconnect standard for heterogeneous computing, satisfies the "Single-Writer-Multiple-Reader" (`SWMR`) property, a key coherence guarantee observed by all cache coherent systems [11].

Our contribution in this paper is not the CXL model and associated proof, which is the subject of a different article [9], but rather a report on the experience of wrestling with a proof at this scale.

The model consists of 68 transition rules, and our proof involved showing that all these rules preserve a property, which we call $\texttt{SWMR}^+$, that implies `SWMR`. $\texttt{SWMR}^+$ is a strengthened version of `SWMR`, consisting of a conjunction of 796 formulas. This amounts to proving 54,128 little lemmas, each showing a certain conjunct i being preserved by rule j. It was not possible to come up with $\texttt{SWMR}^+$ in one go. We started with a first approximation with only 2 conjuncts, and then went through many iterations of refining it, during which the invariant steadily grew in size. These iterations have pushed Isabelle's Prover IDE (`PIDE`) to its limit, creating scalability challenges that have to be addressed via a combination of proof engineering of the theory code, external scripting and reusing and modifying parts of the Isabelle/ML codebase.

Contribution 2: Experience accelerating proof development with `super_sketch`. To cope with the aforementioned challenges, we have developed `super_sketch`, a tool that supports the automatic integration of `sledgehammer`-generated proofs in an Isabelle proof script. The high-level ideas behind `super_sketch` are briefly discussed in a paper devoted to our overall modelling and proof efforts [9]. Our contribution here is a detailed discussion of our experience developing `super_sketch`, and how `super_sketch` is implemented in Isabelle/ML.

Given heuristics supplied by the user, `super_sketch` turns the proof obligations into multiple goals, trying to solve all of them by concurrently calling `sledgehammer`. For more difficult subgoals, `super_sketch` tries to use the extra heuristics to further reduce the subgoals before calling `sledgehammer` on them. We report on our experience leveraging `super_sketch` to automate the task of re-generating proof scripts in each iteration of baking our inductive invariant. We describe the scripting and fine-tuning challenges associated with optimising the usage of `super_sketch`. Towards the end of proof development, the tool was very effective in reducing both the number of iterations we needed to do, and the human effort involved in each iteration.

Contribution 3: A tool that automatically fixes broken proof scripts. `super_sketch` is useful in bulk-generating proof text for a single theorem, but not tackling errors and updates that are interspersed across multiple theorems and files. To address the limitations of `super_sketch`, we have developed `super_fix`, a tool that is better at fixing local errors in an Isabelle proof script than bulk-generation of proof text. This is especially suited for repairing proof errors that arise due to modifications upstream in the proof—e.g. changes to the definitions of our transition system or to the inductive invariant. `super_fix` is good at fixing goal errors and one-liner proof errors, where errors are a minority and interspersed in the proof script. The implementation is inspired by the observation that non-terminating proofs and upstream errors cause later errors and therefore should be fixed first. Using `super_fix`, we have successfully dropped simplifying assumptions we made in obtaining an initial version of the proof of the `SWMR` property, obtaining a stronger theorem.

Our tools, written in Isabelle and standard ML, operate at the outer syntax level. We have made `super_sketch` and `super_fix` publicly available.

II. Background: CXL and the SWMR Property

In this section, we describe the concrete proof engineering problem we needed to address while proving the Single-Writer-Multiple-Reader (SWMR) property of our model of the `CXL.cache` protocol [9].

To contexualise the problem, we briefly introduce cache coherence and Compute Express Link (CXL).

A. CXL and cache coherence

Cache coherence protocols are essential in multi-core systems to ensure all caches share a coherent view of memory. They synchronise multiple copies of the same data among the caches of different cores, preventing incoherent scenarios such as stale data being read by a core that has not been notified about a modified cacheline. Abstractly, a cache coherence protocol can be viewed as a communication protocol over a network interconnecting several cores.

One of the most common cache coherence protocols is the MESI protocol [12]. The MESI acronym refers to four cacheline states: M, indicating that the cacheline has a valid copy of the data and that the data is being modified, so that it must be written back later; E, indicating that the copy is clean and exclusively owned; S, indicating read-only and non-exclusive ownership; and I, indicating that the address is not currently in the cache and therefore is not valid. Each cacheline can transition to other states by sending and receiving certain messages over the communication network, requesting and indicating ownership changes.

CXL is a popular emerging interconnect standard that defines how memory can be shared in a cache-coherent way between heterogeneous devices, such as CPUs, GPUs, and other accelerators. This means that two CXL-enabled devices, even if manufactured as standalone hardware systems, can be composed to present a unified memory and cache system. In `CXL.cache`, a sub-protocol of CXL, these (possibly multicore) devices are abstracted as a single "core" in the larger cache coherent domain, and the CXL interconnect serves as the network for connecting these "cores". `CXL.cache` is a MESI-style, directory-based cache coherence protocol with some clever design choices that make it easier to implement a cache-coherent device.

The `CXL.cache` standard is a suitable candidate for formal verification because it is a relaxed protocol with an unordered network and very few restrictions. This creates complex concurrent situations that are potentially error-prone. Our modelling and proof efforts were worthwhile: they uncovered several inaccuracies in the CXL specification, which have been confirmed by CXL experts. They also revealed fixes for these inaccuracies that are in the process of being adopted [9].

B. An overview of our model

Our model, an operational-style transition system, represents the states and transitions of a `CXL.cache` implementation. A set of system states and rules govern the state transitions. Intuitively, when multiple cache copies have read or write access, these accesses cannot coexist. Otherwise, stale data may be read.

System state representation and transition rules. We define the system state as a record of type `SystemState`, which abstracts relevant components of a `CXL.cache`-enabled cluster of devices: their cachelines, message channels representing the interconnect, and other auxiliary structures that are necessary for CXL-specific restrictions. The details of the datatype can be found in our Isabelle formalisation [13].

Transition rules model the system's possible behaviours and correspond to the protocol atomic actions. We have 68 transition rules, covering all necessary actions to start or complete a coherence transaction. Each transition rule R_i $(1 \leq i \leq 68)$ comprises:

- A guard $\mathtt{guard}_{R_i}$ specifying the conditions under which the rule R_i can be applied.
- A state-updating function f_{R_i} defining how each system state changes when the rule fires.

A system state Σ transitions to a state Σ' (denoted $\Sigma \to \Sigma'$) if there exists a transition rule R in the set $R_1, \ldots, R_{68}$ such that the guard $\mathtt{guard}_R$ holds on Σ, and Σ' is the result of applying the state-update function f_R to Σ. Formally:

$$\Sigma \to \Sigma' \iff \exists R \in \{R_1, \ldots, R_{68}\}. \; \mathtt{guard}_R\, \Sigma \wedge f_R(\Sigma) = \Sigma'$$

The `SWMR` property. The Single-Writer-Muiltiple-Reader (`SWMR`) property is an important coherence guarantee stating that if one device has write permission on a cacheline, no other device simultaneously has read or write permission on that cacheline. Formally:

$$\mathtt{SWMR}\,\Sigma \overset{def}{=} (i \neq j \wedge \Sigma.\mathsf{Cachelines}(\mathsf{Dev}_i) = \mathsf{M} \implies \Sigma.\mathsf{Cachelines}(\mathsf{Dev}_j) \notin \{\mathsf{S}, \mathsf{M}\})$$

Here $\mathsf{Cachelines}(\mathsf{Dev}_i)$ refers to the ith device cacheline. A normal `CXL.cache` device can cache copies of the cachelines from a special device called "Host".

Proof goal. Our goal is to show that starting from any valid initial state, the `SWMR` property holds after any sequence of transitions:

$$\mathtt{InitialState}\,\Sigma \wedge (\Sigma \to^* \Sigma') \implies \mathtt{SWMR}\,\Sigma'.$$

Here, $\to^*$ denotes the reflexive transitive closure of the transition relation $\to$. We need to find an inductive invariant P satisfying the following conditions:

$$\begin{aligned} \mathtt{InitialState}\,\Sigma &\implies P\,\Sigma \\ P\,\Sigma \wedge (\Sigma \to \Sigma') &\implies P\,\Sigma' \\ P\,\Sigma &\implies \mathtt{SWMR}\,\Sigma \end{aligned}$$

Showing that $\mathtt{InitialState}\,\Sigma \implies \mathtt{SWMR}\,\Sigma$ holds is relatively easy, but unfortunately we cannot take `SWMR` as P because it is not inductive. In other words, there are transitions where `SWMR` holds before the transition

$$\begin{pmatrix} \ddots & & \iddots \\ & \begin{pmatrix} P\ \Sigma \wedge \\ \texttt{guard}_{R_i}\ \Sigma \wedge \\ \Sigma \to_{R_i} \Sigma' \\ \implies \\ \phi_j\ \Sigma' \end{pmatrix}_{(i,j)} & \\ \iddots & & \ddots \end{pmatrix}_{m \times n}$$

Fig. 1. Proof obligation matrix for the inductiveness of P. Each single cell represents a certain conjunct being preserved by a rule.

but not after. Consider a transition from Σ to Σ' where a device upgrades its cacheline to the M state while another device already holds the cacheline in the M state:

$$\begin{array}{rlcl} & \Sigma.\mathsf{Cachelines}(\mathsf{Dev}_0) & = & \mathsf{M} \\ \wedge & \Sigma.\mathsf{Cachelines}(\mathsf{Dev}_1) & \neq & \mathsf{M} \\ \wedge & \Sigma'.\mathsf{Cachelines}(\mathsf{Dev}_0) & = & \mathsf{M} \\ \wedge & \Sigma'.\mathsf{Cachelines}(\mathsf{Dev}_1) & = & \mathsf{M} \end{array}$$

Here, assuming two devices, Σ satisifies SWMR, but after the transition to Σ', both device 0 and 1 have their cachelines in the M state, violating SWMR.

We strengthen SWMR by conjoining it with additional properties to form $P = \texttt{SWMR} \wedge \phi_1 \wedge \phi_2 \wedge \ldots$. These additional properties capture the conditions to ensure that SWMR is preserved across all transitions.

The continuously evolving invariant. We start by setting P to SWMR and identify specific scenarios where the invariance property $P\ \Sigma \wedge \Sigma \to \Sigma' \implies P\ \Sigma'$ is violated, using the shorthand notation $\xrightarrow{\tau}$ for transitions leading to such scenarios. We then formulate additional properties ϕ_i to prevent these violations.

$$\Sigma \xrightarrow{\tau} \Sigma' \ \wedge\ \phi_i\ \Sigma \wedge \texttt{SWMR}\ \Sigma \implies \texttt{SWMR}\ \Sigma'$$

But this introduces more proof obligations if ϕ_i is itself not inductive, requiring us to come up with ϕ_{i+1} to ensure that ϕ_i holds in all scenarios:

$$\Sigma \to \Sigma' \ \wedge\ \phi_{i+1}\ \Sigma \implies \phi_i\ \Sigma'$$

This process continues, with each new ϕ_i strengthening P until a fixed point is reached where P is inductive.

The obligation matrix. We can view the task from the perspective of augmenting an $m \times n$ matrix, where m is the number of transition rules, and n is the number of conjuncts in P. Each cell (i, j) in the matrix corresponds to the obligation of proving that ϕ_j is preserved by transition R_i. We illustrate this in Figure 1. Each row and column have a special meaning:

- Rows correspond to individual transition rules.
- Columns correspond to individual conjuncts in P.

We started with a 68×2 matrix (68 rules and 2 conjuncts), and gradually expanded it by adding more conjuncts (so that m remains fixed but n increases as more conjuncts are added). Whenever a cell in the matrix is unprovable, we need to

- Add a new conjunct to P.
- Write the proof to the lemma corresponding to the previously unprovable cell, which is made possible with the new conjunct.
- Write proofs for the additional proof obligations due to the new conjunct being added.

This process repeats until all matrix cells are provable.

III. The Scalability Challenges

We now discuss the scalability challenges we faced during the construction of the formal proof of the SWMR property for our CXL.cache model using Isabelle/HOL. The primary challenge stemmed from the continuously evolving inductive invariant P, which grew significantly in size as we iteratively strengthened it to achieve inductiveness. This growth led to a substantial increase in proof obligations and computational overhead, pushing the capabilities of Isabelle, and the hardware that ran it, to their limits.

Structure of the proofs. Our proof obligations' are organised as in the obligation matrix of Figure 1, which has m rows (transition rules) and n columns (conjuncts of P), as discussed in Section II-B. To manage these obligations effectively, we structured our Isabelle proof files in a "row-major order": each file corresponds to a specific transition rule R_i and contains the rule-related lemmas. This organisation allows us to supply additional facts about specific rules locally to improve the performance of proof automation tools like sledgehammer.

For each transition rule R_i, we aim to prove a lemma that asserts the preservation of the inductive invariant P by that rule. Figure 2 illustrates the typical structure of a lemma and its proof. Each time we add a new conjunct ϕ_{n+1} to the inductive invariant P, we need to:

1) **Add a new fact:** Introduce a new assumption $\mathit{fact}_{n+1} \colon \phi_{n+1}\ \Sigma$.
2) **Add a new goal** Prove that ϕ_{n+1} is preserved by the transition, i.e., show that $\phi_{n+1}(f_{R_i}(\Sigma))$ holds.

These additions are highlighted in blue in Figure 2. We found that the "preamble" section of the proof (highlighted with green) is essential for making our proof scale, even though it might seem redundant or not strictly necessary at first glance. This section is crucial because automated tools like sledgehammer, auto, and simp work significantly better when provided with smaller, focused facts rather than large, complex formulas as a monolithic term like the entire invariant $P\ \Sigma$. Each goal in this proof often depends on only

```
lemma Ri_coherent:
  assumes "P Σ ∧ guard_Ri Σ"
  shows "P (f_Ri(Σ))"
proof -
have fact0:guard_Ri Σ by assumption
have fact1:φ1 Σ by assumption
have fact2:φ2 Σ by assumption
...
have factn:φn Σ by assumption
have factn+1:φn+1 Σ by assumption
show ?thesis
  proof (intro conjI)
    show goal1: "φ1(f_Ri Σ)" Sledgehammer proof
1 using facts from {fact0,...,factn}
  next
    show goal2: "φ2(f_Ri Σ)" Sledgehammer proof
2 using facts from {fact0,...,factn}
  next
    ...
  next
    show goaln: "φn(f_Ri Σ)" Sledgehammer proof
n using facts from {fact0,...,factn}
  next
    show goaln+1: "φn+1(f_Ri Σ)" Sledgehammer
proof n using facts from {fact0,...,factn+1}
  qed
qed
```

Fig. 2. A rule lemma for R_i. Our mechanised proof mainly consists of these rule lemmas. The additions when a new conjunct ϕ_{n+1} is introduced are highlighted in blue.

several facts from $fact_0$ to $fact_n$. Referencing the whole invariant $P\ \Sigma$ is unnecessary and inefficient. Without the "preamble" section that breaks down the invariant P into manageable, digestible individual facts, automated tools like `sledgehammer` would begin to struggle—our experience is that, with more than 100 conjuncts, `sledgehammer` would either fail to find a proof, or find proofs that, when adopted, would lead to non-termination during proof checking.

To amend our proof, it was preferable to add the blue parts (of Figure 2) rather than to regenerate the entire proof of R_i`_coherent`. Verifying whether a proof exists for each newly added goal remained a manual and time-consuming process. The steps involved were:

- Manually copy the new conjunct as a new fact, and add a new proof goal about the new conjunct (the blue bits in fig. 2).
- Manually invoke `sledgehammer` at the position of the new goal.
- Wait for `sledgehammer`'s proof suggestions, which could take up to several minutes per goal.
- Manually adopt one of the suggested proofs into our proof script.
- If `sledgehammer` fails, manually inspect the goal to devise heuristics such as case analysis, simplification, or introduce intermediate lemmas.
- Repeat the steps for all new goals on all rule files.

This process was labour-intensive, as we had to repeatedly copy-and-paste, wait for `sledgehammer` to finish proving each goal, click to adopt the proofs, and manage numerous files. As the number of conjuncts increased beyond 100, this manual overhead became untenable.

Limitations of our initial solutions. As well as using state-of-the-art hardware, we attempted several strategies to save human time and remove redundancy.

We used Python scripts with regular expressions to automate the insertion of new facts and goals. However, this did not eliminate the manual effort required to adopt sledgehammer proofs in each file.

We experimented with different proof script structures to improve processing times. For example, we tried consolidating multiple "have...by..." commands with identical proofs (the "by" part) into a single chained command:

```
have fact0:guard_Ri Σ
and fact1:φ1 Σ
and fact2:φ2 Σ
...
and factn:φn Σ by assumption+
```

where the $+$ operator indicates that a proof method is applied one or more times. However the "`by assumption+`" line at the end of the chain took the prover process of Isabelle an exceedingly long time to interpret, as Isabelle seems to handle the "+" operator super-linearly in this situation.

Despite these efforts, we still hit a scalability wall.

A significant factor contributing to this was the limitations of Isabelle/jEdit, the mandatory interactive interface of Isabelle. Isabelle/jEdit struggled to handle multiple large theory files simultaneously due to their size and complexity. Attempting to import all 60+ rule files at once caused crashes. This instability prevented us from processing the files concurrently, which would have allowed us to adopt `sledgehammer` proofs more efficiently. The sequential nature of our workflow significantly increased the human time required for proof development, as we could not leverage parallelism to expedite the process.

Moreover, processing a single goal within a file could be time-consuming, especially if the goal required additional proof strategies such as case analysis, intermediate lemmas, or simplification with `simp`. It could take several minutes or more to find a proof for one goal.

Another significant factor contributing to the scalability wall was the necessity to delete conjuncts or make changes to the transition system. This was necessary e.g. on receiving comments from industry experts working on the CXL specification, who sometimes clarified how our interpretation of the specification text differed from their intent. When removing a conjunct ϕ_i from the invariant

P, the impact was not confined to the single column corresponding to ϕ_i in our obligation matrix. Since ϕ_i could be referenced in proofs across various lemmas, all cells in the matrix that relied on $fact_i$ needed to be re-examined and updated.

The combination of these factors made the manual approach to proof maintenance impractical as the project scaled, leaving little opportunity to focus on higher-level aspects of the proof.

IV. The `super_sketch` Tool

To address the scalability challenges outlined in Section III, we developed `super_sketch`, a tool designed to automate the generation of proofs with minimal human intervention. We built `super_sketch` upon Haftman's `sketch` [14], which automatically generates an Isar [15] skeleton for a single lemma in Isabelle/HOL. However, `super_sketch` extends this functionality significantly to handle more complex proof strategies and integrate automated proof search tools such as `sledgehammer`. We now present details of `super_sketch`, and explain how it helped to eliminate bottlenecks and allow our proofs to scale.

A. Main features of `super_sketch`

The tool automates the proof generation process by applying various user-specified proof methods and heuristics to each goal. It not only generates the proof skeleton but also attempts to solve each subgoal using a combination of proof tactics, automated provers, and `sledgehammer`.

The `super_sketch` tool allows users to:

1) **Specify initial proof methods**: Apply an initial proof method (e.g. `intro conjI`) to decompose the main goal into subgoals.
2) **Apply additional methods to subgoals**: For all subgoals, specify methods to simplify or manipulate them. This can include tactics like `insert assms`, `cases` and `simp`.
3) **Specify methods to split and reduce complex goals**: Break down complex subgoals into smaller, more manageable sub-subgoals using tactics like `cases` and further simplify them using methods like `auto`.
4) **Invoke multiple instances of `sledgehammer`**: Automatically invoke `sledgehammer` to attempt to automatically prove multiple (sub-)subgoal concurrently.
5) **Quickly identify unprovable goals**: If a goal cannot be proven automatically, `super_sketch` inserts a `sorry` placeholder together with a comment indicating the failed proof attempt, highlighting that manual intervention is required.

B. Workflow of `super_sketch`

A summary for `super_sketch`'s workflow is:

1) **Initial goal processing** First, parse the user-supplied methods. There can be up to four methods, which we denote as `meth1` (the *initial proof* method), `meth2` (the *preprocessing* method), `meth_split` (the *splitting* method) and `meth_reduce` (the *reduction* method). Each method can itself be a composite method, built from multiple child methods using method combinators (such as Isabelle's sequencing operator "`,`").
   Second, apply `meth1` (e.g. `intro conjI`) to decompose the main goal into subgoals.
2) **Concurrent proof text generation from each subgoal** For each subgoal, do the following:
   First, apply the specified preprocessing method `meth2` to the subgoal (e.g. `simp, insert assms`). If this succeeds, return "**apply** `meth2` **done**" as the proof text (meaning that the method `meth2` solves the goal in Isabelle), otherwise proceed to call `sledgehammer`.
   Second, invoke `sledgehammer`. If it succeeds, return the proof text. If it fails, apply the user-specified method `meth_split` to the subgoal (e.g., `cases`) to produce sub-subgoals. Proceed to process each of these sub-subgoals in the next step.
3) **Sub-Subgoal Processing (for failed subgoals)** First, for all sub-subgoals resulting from splitting, apply any specified reduction method `method_reduce`. If all sub-subgoals have been solved, return the text corresponding to all the methods applied so far.
   Second, for each remaining sub-subgoal: try to prove the sub-subgoal using `sledgehammer`. If successful, return `sledgehammer`'s proof text. If not, return text indicating a failure to find a proof.
   Finally, combine all sub-subgoals' returned proofs if they all succeeded. If any of the `sledgehammer` calls failed, use placeholder text to indicate failed proof. Return the combined (or failed) text.
4) **Finalisation** Assemble the proofs of all subgoals (including those with `sorry`) into the Isar skeleton to form the complete proof of the main goal. Then output the generated proof script for adoption.

Sometimes methods that are complementary to and more lightweight than `sledgehammer` can already solve or simplify particular subgoals. For proofs containing many subgoals, it is beneficial to apply these methods before invoking `sledgehammer`. This motivates the inclusion of `meth2` (the *preprocessing* method) in our design.

The *splitting* and *reduction* methods, `meth_split` and `meth_reduce`, are used to tackle harder but still solvable subgoals. If sub-subgoals remain after applying them, `sledgehammer` is invoked on all these sub-subgoals, which can be more computationally intensive than the initial invocation of `sledgehammer`. We have found that the harder yet provable goals usually constitute a small but significant percentage of all subgoals in our use case. Given that processing these sub-subgoals would take at least as much time when done manually, incorporating this heavyweight step is justified.

C. *Example usages of* `super_sketch`

After inserting the `super_sketch` command and it finished running, the proof text in markup format is displayed on the Isabelle/jEdit output panel, which the user can click to adopt. Figure 3 illustrates the process of invoking and adopting the proof text from `super_sketch`, showing the command required to invoke `super_sketch` (top) and the result (bottom).

The text blocks highlighted in pink and purple are newly-generated by `super_sketch`. In this example, $goal_h$ required the processing of sub-subgoals.

With `super_sketch` we were able to generate the vast majority of the proofs for our rule lemmas in under half an hour each, covering over 700 conjuncts. This translates to about one day to iterate through the entire obligation matrix. Towards the end of the development we found that each time we generated the 50,000+ proofs, the number of subgoals for which `super_sketch` failed to find a proof (such that human intervention was required) was less than 100.

Additional applications of `super_sketch` Beyond generating proofs for rule lemmas, `super_sketch` can also facilitate the addition of new conjuncts by defining conjunct lemmas. These lemmas state the proof of an entire column of the obligation matrix, allowing us to generate their proofs in a single step. For instance, if we want to test whether the new proof obligations introduced by adding the formula ϕ_{n+1} to P can be proven, we can invoke `super_sketch` with the same set of arguments as in the previous example. This yields:

lemma ϕ_{n+1}_coherent*:*
  assumes "$P\ \Sigma \wedge \phi_{n+1}$"
  shows "$\bigwedge_{i=1}^{m} \phi_{n+1}\ (f_{R_i}(\Sigma))$"
proof -
have $fact_0 \ldots$
...
show *?thesis*
  proof *(intro conjI)*
    show *goal*$_1$*:* "$\phi_{n+1}(f_{R_1}\Sigma)$" **apply** *(insert assms) Sledgehammer proof 1 using facts from* $\{fact_0, \ldots, fact_n\}$
...
    show *goal*$_m$*:* "$\phi_{n+1}(f_{R_i}\Sigma)$" *Sledgehammer proof n using facts from* $\{fact_0, \ldots, fact_n\}$
  qed
qed

Before:

lemma R_i_coherent*:*
  assumes "$P\ \Sigma \wedge guard_{R_i}\ \Sigma$"
  shows "$P\ (f_{R_i}(\Sigma))$"
proof -
have $fact_0 \ldots$
...
show *?thesis*
super_sketch3 *(intro conjI) (insert assms) (cases "Cachelines(Dev*$_0$*)") (auto)*
qed

After:

lemma R_i_coherent*:*
  assumes "$P\ \Sigma \wedge guard_{R_i}\ \Sigma$"
  shows "$P\ (f_{R_i}(\Sigma))$"
proof -
have $fact_0 \ldots$
...
show *?thesis*
  proof *(intro conjI)*
    show *goal*$_1$*:* "$\phi_1(f_{R_i}\Sigma)$" **apply** *(insert assms) Sledgehammer proof 1 using facts from* $\{fact_0, \ldots, fact_n\}$
    ...
    next
    show *goal*$_h$*:* "$\phi_h(f_{R_i}\Sigma)$" **apply** *(insert assms)* **apply** *(cases "Cachelines(Dev*$_0$*)")* **apply** *(auto) Sledgehammer proofs for sub-subgoals of h using facts from* $\{fact_0, \ldots, fact_n\}$ **done**
    next
    ...
    show *goal*$_k$*:* "$\phi_k(f_{R_i}\Sigma)$" **sorry** *(*failed to find proof in multi-steps*)*
    next
    ...
    show *goal*$_n$*:* "$\phi_n(f_{R_i}\Sigma)$" *Sledgehammer proof n using facts from* $\{fact_0, \ldots, fact_n\}$
  qed
qed

Fig. 3. Proof script before and after invocation of `super_sketch` and adopting `super_sketch`'s generated text

Since the number of proof obligations in a column (m) is smaller than that in a row (n), `super_sketch` takes significantly less time to generate proofs for a conjunct lemma compared to a rule lemma, often completing in minutes rather than tens of minutes. We sometimes batch multiple conjuncts together and attempt to prove them in a single lemma, further reducing the number of iterations and saving human effort.

D. Limitations of super_sketch and mitigations

Despite the significant automation provided by super_sketch, certain aspects prevent it from fully automating the tedious parts of our proof development.

One limitation is that super_sketch occasionally incorporates sledgehammer proofs that result in errors or nontermination during proof-checking when adopted into the proof script. This issue can arise due to discrepancies between the proof context at runtime when sledgehammer is invoked by super_sketch and the proof context in an interactive session using the actual Isar proof text. Ideally these contexts should be identical, but in practice, slight differences can lead to sledgehammer generating "bad proofs" that fail or cause non-termination when used. When manually using sledgehammer, the user can easily select an alternative suggested proof that works. These problematic proofs are not indicative of a soundness bug in sledgehammer; rather, they suggest that a proof does exist, but the particular proof text provided is unsuitable for the goal in its current context.

This issue can be mitigated somewhat by the user breaking down the assumptions of the theorem into smaller named facts. However, this does not completely eliminate the occurrence of broken proofs. This is a problem in our use case, which requires immediately-usable proof text if manual effort is to be avoided.

V. Enhancements with super_fix

Before developing super_fix, we were constrained in the number of iterations we could perform when revising the proof obligation matrix due to limited manpower. During the later phases of our project, the inductive invariant P had still not fully converged, so we set ourselves an initial milestone of getting a meaningful proof completed, even if this required weakening the property being proven slightly.

Specifically, we modified the transition system by adding additional predicates to rule R_i's guard, making it fire in a more restricted set of scenarios and thereby eliminating certain complex concurrent situations from consideration. This adjustment did not alter the overall coherence property but simplified the invariant by reducing the number of cases we needed to handle.

Our modified transition system was identical to the original, except for this strengthened rule. We then proved that all reachable states under this strengthening, from any initial state, satisfy the SWMR property.

Having achieved the milestone, we sought to drop this simplification to obtain the desired full theorem. We were uncertain about the amount of additional human resources required to achieve this by manual effort. Therefore, we concluded that an automated tool addressing the remaining scalability challenges in the proof was necessary to manage our manpower efficiently.

We first identified the remaining automation challenges. The issues of sledgehammer generating invalid proofs or the need to fix broken goals—such as when the transition system or invariant is updated, as described in Section III—can often be efficiently addressed by *local* fixes. By local, we mean fixes that are usually confined within the proof of a single lemma and can be derived from the current proof state.

The key idea of the new tool is to automate the process of fixing a piece of almost-correct theory text in the same way a human user would. By *almost-correct*, we mean that the definitions, functions, and datatypes are all valid—they do not raise error messages. For example, consider an error raised due to a referenced lemma being broken, a **show** statement in an Isar proof failing to refine a goal, or a non-terminating one-liner proof.

A human proof engineer would open the Isabelle/jEdit session on the theory file, scroll down to the point where the error or looping occurs, and attempt to fix it. If the issue can be resolved—for instance, by replacing the proof text with alternative text—a fix is applied; if not, a sorry is inserted to allow the processing to continue. If a goal is incorrect, they would try to update the goal, which is clearly displayed in the proof state.

Our new tool, super_fix, aims to emulate this behavior, automating the process of detecting and fixing such local errors, thereby significantly reducing the manual effort required.

A. Utilising DeepIsaHOL to automate fixes

To implement this procedure, we leverage APIs from the DeepIsaHOL codebase [16] for converting proof scripts into Isabelle's internal representations of proofs. DeepIsaHOL is a project that provides infrastructure for extracting and feeding data to Isabelle. It has a set of APIs to manipulate terms, contexts, transitions, proof states and other Isabelle data structures. These APIs allow us, for example, to easily turn an arbitrary string into the corresponding proof command.

We use DeepIsaHOL APIs to directly access the data-carrying states s of Isabelle's script-checking algorithm. In Isabelle/ML, the type Toplevel.state represents these states. Among other things, they carry theory information (e.g. imported theorems), context information (e.g. current user configuration) and, when proving, a proof's proof states. Isabelle's official constructs for manipulating these states are top-level transitions τ. At the user level, these correspond to the script's commands and their arguments (e.g. **apply** *auto* or **have** "$F\,x = y$"). Intuitively, one can view them as functions f mapping a Toplevel.state s_i to the next one s_{i+1} with optional error messages ε_{i+1} if the transition was not meaningful. Thus, we extensively use DeepIsaHOL's methods to parse a `.thy` file and convert it into a finite sequence $\langle \tau_i \rangle_{i \in I}$ of Isabelle `Toplevel.transitions`. Since the top-level states carry the proof states, if a transition τ_i fails with an error ε_{i+1} inside a proof, we can inspect the error, backtrack, and apply a different and correct transition τ_i' based on the type of error reported.

We mainly focus on three types of errors: non-terminating proofs, goal alignment errors, and incorrect applications of proof methods. We explain their relevance below and our procedures for fixing them.

B. Detecting and handling non-terminating proofs

The type of error that needs to be prioritised is non-terminating or looping proofs, which we use to refer to lines in the proof script that take an indefinite amount of time to process. Visually this is shown on the interactive editor as lines constantly marked with a purple background, indicating that the `PIDE` is not done processing them. Looping proofs are often caused on lines that use automated provers or SMT solvers. For instance, the `auto` prover can loop because of recursive applications of equalities or introduction rules.

Looping errors lead to bottlenecks that prevent the fixing of other errors because processing tools get stuck on them. A human user would resolve a looping error by calling `sledgehammer` at the position of the looping line and adopting a new proof that works. If the user is confident that a proof exists, they may simply declare the subgoal as true with a `sorry` so that they can move on to fix other parts and complete that step later.

We approximate this behaviour programmatically by applying the script's top-level transitions τ_i with timeouts. We sequentially compute the top-level states s_i for each top-level transition τ_i. If τ_i corresponds to a tactic application containing a potentially loop-prone method, we time its application. Then, if the application does not produce a new state after 90 seconds, we replace τ_i with a `sorry`. Either way, we retrieve the new state s_{i+1} and continue processing the script.

We do not fix `sorry`s immediately to ensure progress for upcoming transitions—since `sledgehammer`'s newly-generated proofs may still loop, it may hinder producing any intermediate results. We instead construct a first version $\langle \tau_i \rangle_{i \in I}$ of the fixed proof script with some `sorry` placeholders, and then substitute them with `sledgehammer` proofs in a second pass.

For timing transitions, we take advantage of Isabelle/ML's `Future` library for multi-threaded computations. For a loop-prone transition τ_i, we create a new (Isabelle process) child thread (via `Future.fork`) in charge of applying the transition τ_i to state s_i. From the parent process, we time this child every (OS) 100 milliseconds, and await a result from this computation (of type `'a Future.future`). If there is one (`Future.is_finished`), we extract it (`Future.get_result`), otherwise we cancel the child thread (`Future.cancel`) and report a timeout error, which triggers our algorithm to insert a `sorry`.

C. Handling misaligned proof obligation errors

After a definition modification or an edition of a proof script, some assertions in the proof must change. If this is not addressed, it can reverberate further downstream in the proof, potentially generating infinite loops. Fortunately, Isabelle warns the user when a goal asserted in a transition τ_i does not coincide with the proof obligation obl_i truly required to complete the proof. This is reported in the error ε_{i+1} as: "Failed to refine any pending goal". Notice that the previous state s_i contains the real proof obligation obl_i. Thus, while processing the sequences of transitions $\langle \tau_i \rangle_{i \in I}$, we can detect if the script produces an incorrect transition by inspecting ε_{i+1}. If it does, we extract obl_i from s_i, generate a transition τ_i' with it (e.g. via **show** "obl_i"), and apply it to s_i. Then we can continue processing the original script, and use our other error-correction methods to fix the probably incorrect upcoming method applications.

D. Handling incorrect proof method application

If there is no timeout, but the proof method in a transition τ_i still fails, Isabelle has various messages that could be reported in ε_{i+1}, such as "Illegal application…in state mode" or "Failed to apply proof method". These errors can be consequences of previous fixes from our tool. The first arises when the script attempts a proof method in an already certified step. The second one emerges when the attempted proof method fails and it may arise due to our tools' modification of a proof obligation. In the first case, we simply delete the redundant τ_i. In the second case, we call `sledgehammer` to find a correct proof method. If it does, we replace τ_i with the `sledgehammer`-found one. Otherwise, we write a `sorry`, indicating that the user needs to look into that proof step.

E. Prioritising upstream error fixes

Often lines directly above an error in the proof script were causing the subsequent errors or loops. We want the tool to automatically generate those fixes if possible, and carry on with processing the file as if the fix has been integrated. By prioritizing the repair of root errors, we resolve multiple errors at once.

VI. Manual effort saved: some statistics

While it is impossible to calculate the exact amount of manual effort saved thanks to `super_sketch` and `super_fix`, due to these tools being developed via a gradual, non-uniform process, we summarise with some data points how `super_sketch` and `super_fix` reduced the burden of manual proof maintenance and kept our verification manageable.

A hybrid of scripting and manual effort produced 68 `.thy` files (one per rule) with 777 goals each. After using `super_sketch`, 18 files contained unfinished proofs. Out of these, 13 contained at most 2 unfinished proofs, while the remaining had 14, 10, 6, 6, and 4 errors. Then, after our fix iteration with `super_fix`, only 9 files remained with errors, 1 per each. A second refinement with `super_fix` completed the proof. These `super_fix` numbers are an under-approximation of the total errors fixed since some of the non-terminating proof-error are not recorded in the processing log.

VII. Related Work

Automation tools have been extensively used to enable and accelerate the development of mechanized proofs in various ITPs [17], [18], [19], [20]. In Isabelle/HOL, `sledgehammer` [5], [21], [22], [20], [23], [24] notably integrates automated theorem provers with the proof assistant. This integration has been instrumental in making our work possible. Although other ITPs do not share proof search tools as powerful as `sledgehammer`, the ideas behind `super_sketch` and `super_fix` also serve for proof-engineering automation in other ITPs. Large parts of the capabilities of `super_sketch` do not rely on `sledgehammer` but rather a solver that can automatically discharge simple goals. Any automation utilities in other ITPs can be encapsulated as we have done for `simp` and `auto`.

There are other tools that do not directly rely on external provers or SMT solvers, improve the proof-engineering efficiency, and automate tedious proof-maintenance tasks. Eisbach [25] is an Isabelle-based proof method language that allows users to build complex proof methods from simpler ones, supporting abstraction, recursion, and pattern matching. Matichuk et al. [26] have demonstrated that applying Eisbach can reduce proof script sizes to a fraction of their original implementation in certain sections of practical formalisations such as seL4 [6]. Eisbach could reduce code duplication in our formalisation by encapsulating frequently used compound proof methods leading to better proof automation in our work.

Smart_Isabelle [27], [28], [29], [30] is a suite of tools that leverages `sledgehammer` and other automated provers to find proofs for harder theorems than a single `sledgehammer` call can handle. These tools use clever heuristics that exploit the syntactic structure of problems, especially for induction problems. Its most resource-intensive tool, `tryhard`, provides Isabelle users with a command that generates proof text by automatically searching through multiple intermediate steps. During the development of our formalisation, we employed `tryhard` as a stronger alternative to `sledgehammer`, which generates proofs for a single subgoal similar to the sub-subgoal processing triggered in `super_sketch`. However, `tryhard` proved to be overly resource-intensive and therefore unsuitable for the number of subgoals in our use case. Nevertheless, it inspired us to develop the sub-subgoal processing step in our `super_sketch` tool.

Controlled automation [31] is a tactic developed in HOL4 [32] to enhance the productivity of proof engineers and reduce the verbosity of proof scripts. It allows the user to provide minimal guidance to the prover via a mechanism called hints, and precisely control the modifications to assumptions and goals.

We have chosen to use Isabelle for its expressivity, flexibility and small trusted kernel. Our initial conjecture was that starting with two devices would allow us to scale better and obtain quick initial results, which we could generalise to arbitrary many devices. In ongoing work we are investigating this parametrisation. It would be interesting to export the model (in its two-device form) to more automated tools like IC3 [33], Murphi [34] and IVy [35] and compare the results.

VIII. Future Work

We wish to make `super_sketch` more generally applicable by enabling the automatic usage of proof methods such as term-accepting induction tactics. This enhancement would allow users to avoid manually inputting heuristics, specially for induction proofs.

We also aim to extend `super_fix`'s capabilities by incorporating more sophisticated proof repair techniques [19]. This would allow it to fix proofs considerably different from their previous iterations.

Incorporating all our tools and scripts into a single and fully automated pipeline would improve the overall approach. A tighter integration of solvers (possibly bypassing `sledgehammer`) with the ITP is needed to make such a pipeline efficient for large verifications.

Finally, we intend to extend our CXL model to support more devices and memory locations. It currently has three devices and a single memory location. While this is sufficient for verifying cache coherence, supporting more devices and locations is essential for tasks such as litmus testing and other memory consistency verification tasks. We expect to reuse the existing proof infrastructure, adapting the proofs to newer versions with more components, using `super_sketch` and `super_fix` to iterate and progress with less human effort.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work was supported by EPSRC grant EP/R006865/1 and a Horizon MCSA 2022 Postdoctoral Fellowship (project number 101102608).

References

[1] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283. [Online]. Available: https://doi.org/10.1007/3-540-45949-9

[2] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. [Online]. Available: https://doi.org/10.1007/978-3-662-07964-5

[3] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The Lean theorem prover (system description)," in *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, ser. Lecture Notes in Computer Science, A. P. Felty and A. Middeldorp, Eds., vol. 9195. Springer, 2015, pp. 378–388. [Online]. Available: https://doi.org/10.1007/978-3-319-21401-6_26

[4] Agda Developers, "Agda," accessed: 2025-02-03. [Online]. Available: https://agda.readthedocs.io/

[5] J. C. Blanchette, S. Böhme, and L. C. Paulson, "Extending Sledgehammer with SMT solvers," *J. Autom. Reason.*, vol. 51, no. 1, pp. 109–128, 2013. [Online]. Available: https://doi.org/10.1007/s10817-013-9278-5

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. A. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel." in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[7] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[8] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: a verified implementation of ML," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 179–192. [Online]. Available: https://doi.org/10.1145/2535838.2535841

[9] C. Tan, A. F. Donaldson, and J. Wickerson, "Formalising CXL cache coherence," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2025*. ACM, 2025, to appear.

[10] CXL Consortium, "Compute Express Link Specification, Revision 3.1," 2023, accessed: 2025-02-03. [Online]. Available: https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf

[11] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020. [Online]. Available: https://doi.org/10.2200/S00962ED2V01Y201910CAC049

[12] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*, D. P. Agrawal, Ed. ACM, 1984, pp. 348–354. [Online]. Available: https://doi.org/10.1145/800015.808204

[13] C. Tan, "CXLCoherenceProof GitHub repository," 2023, accessed: 2025-02-04. [Online]. Available: https://github.com/ChengsongTan/CXLCoherenceProof

[14] F. Haftmann, "The Sketch and Explore library," 2023, accessed: 2025-02-04. [Online]. Available: https://isabelle.in.tum.de/dist/library/HOL/HOL-ex/Sketch_and_Explore.html

[15] M. Wenzel and L. C. Paulson, "Isabelle/Isar," in *The Seventeen Provers of the World, Foreword by Dana S. Scott*, ser. Lecture Notes in Computer Science, F. Wiedijk, Ed. Springer, 2006, vol. 3600, pp. 41–49. [Online]. Available: https://doi.org/10.1007/11542384_8

[16] J. J. Huerta y Munive, "DeepIsaHOL," Nov. 2023, accessed: 2025-02-04. [Online]. Available: https://github.com/yonoteam/DeepIsaHOL

[17] LeanProver Community, "Lean-auto," 2024, accessed: 2025-02-04. [Online]. Available: https://github.com/leanprover-community/lean-auto

[18] F. Lindblad and M. Benke, "A tool for automated theorem proving in Agda," in *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. Filliâtre, C. Paulin-Mohring, and B. Werner, Eds., vol. 3839. Springer, 2004, pp. 154–169. [Online]. Available: https://doi.org/10.1007/11617990_10

[19] T. Ringer, R. Porter, N. Yazdani, J. Leo, and D. Grossman, "Proof repair across type equivalences," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 112–127. [Online]. Available: https://doi.org/10.1145/3453483.3454033

[20] L. C. Paulson and K. W. Susanto, "Source-level proof reconstruction for interactive theorem proving," in *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, ser. Lecture Notes in Computer Science, K. Schneider and J. Brandt, Eds., vol. 4732. Springer, 2007, pp. 232–245. [Online]. Available: https://doi.org/10.1007/978-3-540-74591-4_18

[21] J. Meng and L. C. Paulson, "Experiments on supporting interactive proof using resolution," in *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, ser. Lecture Notes in Computer Science, D. A. Basin and M. Rusinowitch, Eds., vol. 3097. Springer, 2004, pp. 372–384. [Online]. Available: https://doi.org/10.1007/978-3-540-25984-8_28

[22] J. Meng, C. Quigley, and L. C. Paulson, "Automation for interactive proof: First prototype," *Inf. Comput.*, vol. 204, no. 10, pp. 1575–1596, 2006. [Online]. Available: https://doi.org/10.1016/j.ic.2005.05.010

[23] J. Meng and L. C. Paulson, "Lightweight relevance filtering for machine-generated resolution problems," *J. Appl. Log.*, vol. 7, no. 1, pp. 41–57, 2009. [Online]. Available: https://doi.org/10.1016/j.jal.2007.07.004

[24] ——, "Translating higher-order clauses to first-order clauses," *J. Autom. Reason.*, vol. 40, no. 1, pp. 35–60, 2008. [Online]. Available: https://doi.org/10.1007/s10817-007-9085-y

[25] D. Matichuk, M. Wenzel, and T. C. Murray, "An isabelle proof method language," in *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, ser. Lecture Notes in Computer Science, G. Klein and R. Gamboa, Eds., vol. 8558. Springer, 2014, pp. 390–405. [Online]. Available: https://doi.org/10.1007/978-3-319-08970-6_25

[26] D. Matichuk, T. C. Murray, and M. Wenzel, "Eisbach: A proof method language for Isabelle," *J. Autom. Reason.*, vol. 56, no. 3, pp. 261–282, 2016. [Online]. Available: https://doi.org/10.1007/s10817-015-9360-2

[27] Y. Nagashima, "Faster smarter proof by induction in Isabelle/HOL," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Z. Zhou, Ed., 2021, pp. 1981–1988. [Online]. Available: https://doi.org/10.24963/ijcai.2021/273

[28] ——, "SeLFiE: Modular semantic reasoning for induction in Isabelle/HOL," *CoRR*, vol. abs/2010.10296, 2020. [Online]. Available: https://arxiv.org/abs/2010.10296

[29] ——, "Smart induction for Isabelle/HOL (tool paper)," in *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020,

pp. 245–254. [Online]. Available: https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_32

[30] ——, “Towards united reasoning for automatic induction in Isabelle/HOL,” in *The Japanese Society for Artificial Intelligence 34th Annual Conference (JSAI), online*, 2020. [Online]. Available: https://doi.org/10.11517/pjsai.JSAI2020.0_3G1ES103

[31] E. Kang and M. D. Aagaard, “Improving the usability of HOL through controlled automation tactics,” in *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, ser. Lecture Notes in Computer Science, K. Schneider and J. Brandt, Eds., vol. 4732. Springer, 2007, pp. 157–172. [Online]. Available: https://doi.org/10.1007/978-3-540-74591-4_13

[32] HOL4 Development Team, “HOL4,” 2024, accessed: 2025-01-27. [Online]. Available: https://hol-theorem-prover.org

[33] A. R. Bradley, “SAT-based model checking without unrolling,” in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 70–87. [Online]. Available: https://doi.org/10.1007/978-3-642-18275-4_7

[34] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, “Protocol verification as a hardware design aid,” in *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computer & Processors, ICCD ’92, Cambridge, MA, USA, October 11-14, 1992*. IEEE Computer Society, 1992, pp. 522–525. [Online]. Available: https://doi.org/10.1109/ICCD.1992.276232

[35] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: safety verification by interactive generalization,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. D. Berger, Eds. ACM, 2016, pp. 614–630. [Online]. Available: https://doi.org/10.1145/2908080.2908118