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

Anonymous Author(s)
Anonymous Affiliation(s)
Email: anonymous@domain.com

***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 hundreds 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 the challenges in proof engineering this, 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 discuss the iterative development of the inductive invariant, the lack of modularity in proofs of inductive invariants, and how `super_sketch` mitigates these issues. 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 definition changes. This allowed us to drop simplifying restrictions in our model whilst retaining the correctness proof, as we avoid significant manual effort inspecting and fixing the broken lines of the original proof when making the model more general. 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*—Cache Coherence, Mechanised Proof, Isabelle, CXL, SWMR, Proof Engineering, Proof Automation**

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 users' theorems automatically. Despite `sledgehammer`'s usefulness, the user still needs to wait for a long time (tens of seconds, sometimes minutes or even more if the timeout is set to be larger) 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 of them. 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, 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 invariably 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. Eventually, our proof converged but, in the process, the number of proof goals ballooned, leading to each iteration of the process taking a large amount of human effort and machine time.

We found that performing a mechanised proof at this

scale using standard tools became infeasible. We used a high amount of machine resources (and wall clock time) waiting for `sledgehammer` to prove subgoals that it had repeatedly proven during previous 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 various 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 thousand proof goals.

We expect that details of our proof effort, together with the tooling we have designed to cope with scalability challenges, will be interesting and useful to others who are embarking on large mechanised proofs, hence this experience report.

Contribution 1: The challenges of working on a large-scale mechanised proof about the correctness of emerging hardware. Over the last two years, we have developed a formal proof in Isabelle/HOL consisting of 74 Isabelle theory files, totalling around 310k lines of code. The proof is about a model we built for CXL (Compute Express Link [9]), an important industry interconnect standard for heterogeneous computing. Our aim has been to show that the model satisfies the "Single-Writer-Multiple-Reader" (`SWMR`) property, an important coherence guarantee observed by all cache coherent systems [10].

Our contribution in this paper is not the model of CXL and associated proof, but rather a report on the experience of wrestling with a proof at this scale. An article about the model, containing cache coherence-specific technical details of the proof and targeted at the systems community, has been published as a technical report which we do not cite here to avoid violating the double-blind review process.

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 implementing `super_sketch` to accelerate proof development. 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 introduced in the above-mentioned technical report. 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 our user 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 to optimise the usage of `super_sketch`. Towards the end of the proof development, the tool has been very effective in both reducing the number of iterations we needed to do and also reducing the human effort in starting a new iteration. We provide statistics about how much manual effort has been saved.

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 "proof repairing" errors due to modifications that happen upstream to the proof–e.g. changes to the definitions of our transition system and the inductive invariant itself. `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.

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.

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 different cores' caches, preventing incoherent scenarios such as stale data being read by a core not notified with a modified cacheline. Abstractly, a cache coherence protocol can be viewed as a communication protocol over a network interconnecting several cores.

CXL is a popular emerging interconnect that allows cache-coherent memory sharing 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 serve as the network for connecting these "cores".

`CXL.cache` is a suitable candidate for formal verification because it is a relaxed protocol with an unordered network and very few restrictions, giving rise to complex concurrent situations that are potentially error-prone. Our proof efforts were indeed worthwhile, as we uncovered several inaccuracies in the CXL specification, which have been confirmed by CXL experts and are in the process of being adopted.

B. An overview of our model

Our model is an operational-style transition system representing the states and transitions of a system implementing `CXL.cache`. We define a set for system states and transitions between these states.

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.

The possible behaviours of the system are modelled by transition rules, each representing an atomic action in the protocol. Each transition rule R_i comprises:

- A guard $\texttt{guard}_{R_i}$ 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 T transitions to a state T' (denoted $T \to T'$) if there exists a transition rule R in the set $R_1, \ldots, R_{68}$ such that the guard $\texttt{guard}_R$ holds on T, and T' is the result of applying the state-update function f_R to T. Formally:

$$T \to T' \iff \exists R \in \{R_1, \ldots, R_{68}\}. \; \texttt{guard}_R\, T \wedge f_R(T) = T'$$

We have 68 transition rules, covering all necessary actions to start or complete a coherence transaction.

The `SWMR` property. The Single-Writer-Muiltiple-Reader (`SWMR`) property is an important coherence guarantee stating that no two devices can have write+write or write+read permission on the same cacheline simultaneously. Formally, we define `SWMR` as:

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

Here $\mathsf{Cachelines(Dev)}_i$ refers to the cacheline of a device i. A normal `CXL.cache` device is allowed to 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:

$$\texttt{InitialState}(T) \wedge T \to^* T' \implies \texttt{SWMR}\, T.$$

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

$$\begin{aligned} \texttt{InitialState}(T) &\implies P\, T \\ P\, T \wedge T \to T' &\implies P\, T' \\ P\, T &\implies \texttt{SWMR}\, T \end{aligned}$$

It is relatively easy to show that $\texttt{InitialState}(T) \implies \texttt{SWMR}\ T$ holds, 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 but not after. Consider a transition from T to T' where a device upgrades its cacheline to the M state while another device already holds the M state:

$$T.\mathsf{Cachelines(Dev)}_0 = \mathsf{M}, \quad T.\mathsf{Cachelines(Dev)}_1 \neq \mathsf{M}, \quad T'.\mathsf{Cachelines(Dev)}_1 = \mathsf{M}$$

Here, T satisifies `SWMR`, but after the transition to T', both device 0 and 1 have their cachelines in the Modified 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.

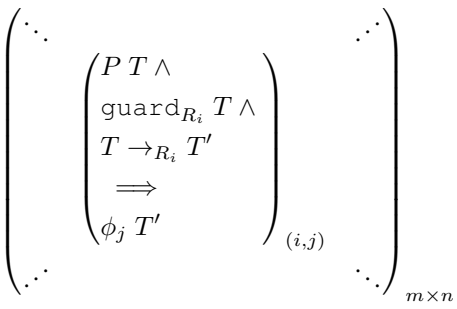

Fig. 1. Proof obligation matrix for the inductiveness of P. Each single cell represents a certain conjunct being preserved by a rule.

The continuously evolving invariant. We start by setting P to `SWMR` and identify specific scenarios where the invariant property $P\ T \wedge T \rightarrow T' \implies P\ T'$ 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.

$$T \xrightarrow{\tau} T' \ \wedge\ \phi_i\ T \wedge \texttt{SWMR}\ T \implies \texttt{SWMR}\ T'$$

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:

$$T \rightarrow T' \ \wedge\ \phi_{i+1}\ T \implies \phi_i\ T'$$

This process continues, with each new ϕ_i added to strengthen P until we reach a fixed point where P is indeed inductive.

The obligation matrix. We can view the task from the perspective of augmenting an $m \times n$, 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. Whenever a cell in that 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 is repeated until we achieve our goal–all cells in the matrix are provable.

III. The Scalability Challenges

In this section, we 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 on which we executed the prover, to their limits.

Structure of the proofs. Our proof obligations are organised based on the obligation matrix in Figure 1, which has m rows (transition rules) and n columns (conjuncts of P). To manage these obligations effectively, we structured our Isabelle proof files in a "row-major order", meaning that each file corresponds to a specific transition rule R_i and contains the lemmas related to that rule. 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 $\mathit{fact}_{n+1}\colon \phi_{n+1}\ T$.
2) **Add a new goal** Prove that ϕ_{n+1} is preserved by the transition, i.e., show that $\phi_{n+1}(f_{R_i}(T))$ holds.

These additions are highlighted in blue in Figure 2. We found that the "preamble" section of the proof (which we highlighted with green) is essential at 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 $P\ T$. Each goal in this proof often depends on only several facts from fact_0 to fact_n. Referencing the whole invariant $P\ T$ is both unnecessary and inefficient. Without the "preamble" section that breaks down the invariant P into manageable, digestible individual facts, automated tools like `sledgehammer` would struggle–for over 100 conjuncts, `sledgehammer` would either fail to find a proof or find proofs such that when adopted, the process of checking them do not terminate.

To amend our proof, it is more sensible to add the blue parts in the proof rather than regenerate the entire proof of R_i`_coherent`. Verifying whether a proof exists for

```
lemma R_i_coherent:
  assumes "P T ∧ guard_{R_i} T"
  shows "P (f_{R_i}(T))"
proof −
have fact_0:guard_{R_i} T by assumption
have fact_1:φ_1 T by assumption
have fact_2:φ_2 T by assumption
...
have fact_n:φ_n T by assumption
have fact_{n+1}:φ_{n+1} T by assumption
show ?thesis
  proof (intro conjI)
    show goal_1: "φ_1(f_{R_i}T)" Sledgehammer proof
1 using facts from {fact_0,...,fact_n}
  next
    show goal_2: "φ_2(f_{R_i}T)" Sledgehammer proof
2 using facts from {fact_0,...,fact_n}
  next
    ...
  next
    show goal_n: "φ_n(f_{R_i}T)" Sledgehammer proof
n using facts from {fact_0,...,fact_n}
  next
    show goal_{n+1}: "φ_{n+1}(f_{R_i}T)" Sledgehammer
proof n using facts from {fact_0,...,fact_{n+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.

each newly added goal remained a manual and time-consuming process. The steps involved were:

- Manually copy the new conjunct to add it as the new fact, and copy from the proof state to add it as the new goal.
- Manually invoke `sledgehammer` at the position of the new goal.
- Wait for `sledgehammer` to generate 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 to find a proof, manually inspect the goal to devise heuristics such as case analysis, simplification, or introduce intermediate lemmas.
- Repeat this process for all new goals across 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. 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 considered consolidating the proofs to the facts into a single command, where the $+$ operator indicates that a proof method is applied one or more times:

  ```
  have fact_0:guard_{R_i} T
  and fact_1:φ_1 T
  and fact_2:φ_2 T
  ...
  and fact_n:φ_n T by assumption+
  ```

  However the "`by assumption+`" line would not terminate with just a few dozen facts, as Isabelle seems to handle the "+" operator super-linearly in this use case.
- We tried hardware upgrades by using machines with larger memory, for example, machines with 64 GB RAM, which improved proof checking time, but did not resolve the underlying workflow bottlenecks.

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. During this time, the user was occupied, waiting for `sledgehammer` to attempt to find a proof, often with little to do but monitor the progress.

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. `super_sketch` builds upon Haftman's `sketch` [11], which automatically generates an Isar [12] 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 like `sledgehammer`. As mentioned earlier, the tool has been briefly introduced in our technical report describing our CXL modelling efforts, and our contribution here is to present the tool in detail and show how it eliminated the bottlenecks in scaling of our proofs.

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 each (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`

The workflow of `super_sketch` can be summarised as follows:

1) **Initial goal processing** First, parse the user-supplied methods. There can be up to four methods, which we denote as `meth1`, `meth2`, `meth_split` and `meth_reduce`. Each method can itself be a composite method, built from multiple children methods using the method combinators (such as Isabelle's sequencing operator "`,`").
   Second, apply the initial proof method `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 already succeeds, return "**apply** `meth2` **done**" (which means applying the method `meth2` solves the goal in Isabelle) as the proof text, otherwise proceed to call `sledgehammer`.
   Second, invoke `sledgehammer`. If it succeeds, return the proof text. If it fails, apply the user-specified splitting 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: Attempt to prove the sub-subgoal using `sledgehammer`. If `sledgehammer` succeeds, return the proof text. Otherwise, return text indicating failure to find a proof.
   Finally, combine the returned proofs of sub-subgoals 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 proof skeleton to form the complete proof of the main goal. Then output the generated proof script for adoption.

Some methods that are complementary to and more lightweight than `sledgehammer` can already solve or greatly simplify the subgoal. For proofs containing many subgoals, it is beneficial to apply these methods before invoking `sledgehammer`. This motivates the inclusion of `meth2` in our design.

The method `meth_split` and `meth_reduce` are used to tackle harder but still solvable subgoals. If sub-

Before:

```
lemma R_i_coherent:
  assumes "P T ∧ guard_{R_i} T"
  shows "P (f_{R_i}(T))"
proof -
have fact_0 ...
...
show ?thesis
super_sketch3 (intro conjI) (insert
assms) (cases "Cachelines(Dev)_0")
(auto)
qed
```

Fig. 3. Proof script before invoking `super_sketch`

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`. 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`

`super_sketch` produces the proof text in markup format on the Isabelle/jEdit output panel, which the user can click to adopt.

Figures 3 and 4 illustrate the process of invoking and adopting the proof text from `super_sketch`, with Figure 3 showing the command required to invoke `super_sketch`, and Figure 4 the result.

The text blocks highlighted in pink and purple in Figure 4 are newly-generated from `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, each time we regenerate all the over 50,000 proofs, only less than 100 subgoals need human intervention (failed to find proof with `super_sketch`).

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

After:

```
lemma R_i_coherent:
  assumes "P T ∧ guard_{R_i} T"
  shows "P (f_{R_i}(T))"
proof -
have fact_0 ...
...
show ?thesis
  proof (intro conjI)
    show goal_1 : "φ_1(f_{R_i}T)" apply(insert
assms) Sledgehammer proof 1 using
facts from {fact_0,...,fact_n}
    ...
    next
    show goal_h : "φ_h(f_{R_i}T)" apply(insert
assms) apply (cases "Cachelines(Dev)_0")
apply (auto) Sledgehammer proofs for
sub-subgoals of h using facts from
{fact_0,...,fact_n} done
    next
    ...
    show goal_k : "φ_k(f_{R_i}T)" sorry (*failed
to find proof in multi-steps*)
    next
    ...
    show goal_n : "φ_n(f_{R_i}T)" Sledgehammer
proof n using facts from
{fact_0,...,fact_n}
  qed
qed
```

Fig. 4. After invocation of `super_sketch` and adopting `super_sketch`'s generated text

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 T ∧ φ_{n+1}"
  shows "⋀_{i=1}^m φ_{n+1} (f_{R_i}(T))"
proof -
have fact_0 ...
...
show ?thesis
   proof (intro conjI)
     show goal_1 : "φ_{n+1}(f_{R_1}T)" apply(insert
assms) Sledgehammer proof 1 using facts
from {fact_0,...,fact_n}
...
    show goal_m : "φ_{n+1}(f_{R_i}T)" Sledgehammer
proof n using facts from {fact_0,...,fact_n}
  qed
qed
```

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.

However, despite these advantages, `super_sketch` has limitations that still require a moderate amount of human effort to generate usable proof text.

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.

To mitigate this issue when using `super_sketch`, we advise users to break down the assumptions of the theorem into smaller, named facts. This approach seems to improve the success rate of valid `sledgehammer` proofs not just in our rule lemmas but also in theorems involving much smaller definitions, although the exact reason is unclear.

However, these mitigations do not completely eliminate the occurrence of broken proofs. In our use case, this is particularly problematic because we require not only confirmation that the proofs are valid but also immediately usable proof text. Even if only a small percentage of the proofs are problematic, fixing them manually would require several workdays—a significant hindrance to productivity.

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. Towards the later phase of our project, the inductive invariant P had still not fully converged, so we have 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 the guard of rule R_i, 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, starting from any initial state, all states reachable under this strengthened transition system satisfy the `SWMR` property.

Having achieved the milestone, we sought to drop this simplification to obtain the desired 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 a new one—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 Isabelle/ML library functions for converting proof scripts into Isabelle's internal representations of proofs using APIs from the DeepIsaHOL codebase [13]. 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 access directly 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 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 are the bottlenecks of fixing other errors because they cause processing tools to be stuck on them. A human user would unstuck this 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 t 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 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. Related Work

Proof automation tools have been extensively used to enable and accelerate the development of mechanized proofs in various interactive theorem provers [14], [15], [16], [17]. In Isabelle/HOL, significant efforts have been made to integrate automated theorem provers with the proof assistant, notably through the development of `sledgehammer` [5], [18], [19], [17], [20], [21]. This integration has been instrumental in making the work presented in this paper possible.

Other tools that do not directly rely on external provers or SMT solvers have also been developed to improve the efficiency of proof engineering and automate tedious proof maintenance tasks.

Eisbach [22] is a proof method language in Isabelle that allows users to build complex proof methods from simpler ones, supporting abstraction, recursion, and pattern matching. Matichuk et al. [6] have demonstrated that applying Eisbach can reduce proof script sizes to a fraction of their original implementation in practical formalizations like seL4. Eisbach has the potential to reduce code duplication in our formalization by encapsulating frequently used compound proof methods using the method definition mechanism. Exploring more advanced uses of Eisbach could lead to better proof automation in our work.

Smart_Isabelle [23], [24], [25], [26] 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 formalization, 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.

VII. Future Work

There are several avenues for future development of both the `super_sketch` and `super_fix` tools, as well as enhancements to the underlying project that led to their creation.

For `super_sketch`, we plan to make the tool more generally applicable by enabling the automatic generation of proof methods. This enhancement would allow users to avoid manually inputting heuristics when using `super_sketch` and enable tactics that work on inductive variables inside specific constructor patterns (e.g., for a lemma on a list l, devise methods that refer to the variable as in the inductive case $l = [a :: as]$). This capability would facilitate greater automation for theorems involving induction on inductive datatypes.

For `super_fix`, we aim to extend its utility by incorporating more sophisticated proof repair techniques, such as those developed in [16]. This would enhance its ability to automatically fix proofs that are taking a bigger step from their old versions.

`super_sketch` and `super_fix` have enabled us to improve our current formalisation significantly. We plan to extend the model to support more devices and memory locations. Currently, our model supports only three devices (a special device called Host in CXL terms, and two caching devices) and a single memory location. While this is sufficient for our specific purpose of verifying cache coherence, supporting more devices and locations is essential for tasks such as litmus testing and other memory consistency verification tasks. We aim 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 reasonable human effort.

Furthermore, we benefited from the third-party library DeepIsaHOL to interface with the internals of Isabelle/ML so that Isabelle's parsers and proof checkers can be easily leveraged with a friendly programming interface. We wish to extend the infrastructure of DeepIsaHOL to make it available and useful for better proof automation in a wider variety of formalisations.

References

[1] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[2] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[3] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *Automated Deduction - CADE-25*, A. P. Felty and A. Middeldorp, Eds. Cham: Springer International Publishing, 2015, pp. 378–388.

[4] Agda Developers, "Agda." [Online]. Available: https://agda.readthedocs.io/

[5] J. C. Blanchette, S. Böhme, and L. C. Paulson, "Extending sledgehammer with smt solvers," in *Automated Deduction – CADE-23*, N. Bjørner and V. Sofronie-Stokkermans, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 116–130.

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. 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 ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 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, p. 107–115, Jul. 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," *SIGPLAN Not.*, vol. 49, no. 1, p. 179–191, Jan. 2014. [Online]. Available: https://doi.org/10.1145/2578855.2535841

[9] CXL Consortium, "Compute Express Link Specification, Revision 3.1," 2023, https://bit.ly/cxl31.

[10] V. Nagarajan, D. Sorin, M. Hill, and D. Wood, *A Primer on Memory Consistency and Cache Coherence, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, Feb. 2020.

[11] F. Haftmann, "The Sketch and Explore library," 2023, https://isabelle.in.tum.de/dist/library/HOL/HOL-ex/Sketch_and_Explore.html.

[12] M. Wenzel and L. Paulson, *Isabelle/Isar*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 41–49. [Online]. Available: https://doi.org/10.1007/11542384_8

[13] J. J. Huerta y Munive, "DeepIsaHOL," Nov. 2023. [Online]. Available: https://github.com/yonoteam/DeepIsaHOL

[14] L. Community, "lean-auto: Automated reasoning in lean," https://github.com/leanprover-community/lean-auto, 2024, accessed: 2024-11-25.

[15] F. Lindblad and M. Benke, "A tool for automated theorem proving in agda," in *Types for Proofs and Programs*, J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 154–169.

[16] 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

[17] L. C. Paulson and K. W. Susanto, "Source-level proof reconstruction for interactive theorem proving," in *Theorem Proving in Higher Order Logics*, K. Schneider and J. Brandt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 232–245.

[18] J. Meng and L. Paulson, "Experiments on supporting interactive proof using resolution," vol. 3097, 07 2004, pp. 372–384.

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

[20] J. Meng and L. C. Paulson, "Lightweight relevance filtering for machine-generated resolution problems," *Journal of Applied Logic*, vol. 7, no. 1, pp. 41–57, 2009, special Issue: Empirically Successful Computerized Reasoning. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570868307000626

[21] ——, "Translating higher-order clauses to first-order clauses," *Journal of Automated Reasoning*, vol. 40, no. 1, pp. 35–60, 2008. [Online]. Available: https://doi.org/10.1007/s10817-007-9085-y

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

[23] Y. Nagashima, "Faster smarter proof by induction in isabelle/hol," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Z.-H. Zhou, Ed. International Joint Conferences on Artificial Intelligence Organization, 8 2021, pp. 1981–1988, main Track. [Online]. Available: https://doi.org/10.24963/ijcai.2021/273

[24] ——, "Selfie: Modular semantic reasoning for induction in isabelle/hol," *ArXiv*, vol. abs/2010.10296, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:224803100

[25] ——, "Smart induction for isabelle/hol (tool paper)," in *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2020.

[26] ——, "Towards united reasoning for automatic induction in isabelle/hol," in *The Japanese Society for Artificial Intelligence 34th Annual Conference (JSAI), online*, vol. https://doi.org/10.11517/pjsai.JSAI2020.0_3G1ES103, 2020.