Automated Sketching and Repairing of Large Mechanised Proofs

Chengsong Tan*†, Alastair F. Donaldson*, John Wickerson*, and Jonathan Julián Huerta y Munive‡
*Imperial College London, UK. Email: {alastair.donaldson, j.wickerson}@imperial.ac.uk
†Kaihong, China. Email: tanchengsong@kaihong.com
‡Aalborg University, Denmark. Email: jjhymuni@cs.aau.dk

This extended abstract summarises ongoing extensions to a conference paper [1] about a suite of proof automation tools that enable rapid iterations on large mechanised proofs. The initial work aimed to address the pain points of proof engineering with ITPs, where users need to specify properties and refine them multiple times to attain their correctness. Specifically, the target is a workflow, where a proof engineer provides an initial draft of a theorem (a conjecture), uses automated proving tools to either confirm the theorem's assertions or identify the error in the specification, with the objective of providing a new draft. Automated tooling that minimises human labour in this process can sometimes determine whether a proof is finished or not.

The use case that motivated this work stems from the real-world example of developing a formal model of CXL.cache in Isabelle/HOL. We specified a Single-Writer-Multiple-Reader (SWMR) property whose proof aided us in identifying bugs in the protocol and in establishing cache coherence of our model.

We omit the details of the background settings of the proof, but the main theorem we are interested in generally looks like the matrix in fig. 1. Namely, we need to specify a property P that is *preserved* by all transitions of the cache coherence system. That is, for each transition $\sigma_1 \rightarrow_{R_i} \sigma_2$, if P holds for the starting state σ_1, then P must hold for the ending state σ_2. Since the property P is a conjunction of smaller (atomic) formulas, we have a matrix of subgoals where a cell at position (i, j) denotes rule number i is preserved by conjunct number j.

The property P is an *inductive* invariant of our cache coherence system because if P is preserved for a single step, then we need to show by induction that it holds for all states taking arbitrarily many steps from the initial state satisfying P. The process of turning our initial specification into an inductive invariant went beyond our original expectations. We started with the SWMR property, which was composed of only 2 conjuncts. With around 70 transition rules, we thought it would be a moderate proof. All we needed to do was to check whether the $70 * 2$ subgoals could be proven. This was supposed to be doable both manually or using automation such as Sledgehammer. However, we soon entered into a loop where we realised that a small set of the conjuncts in the property was not preserved by some rules. This realisation required us to add conjuncts to P or modify the transition system itself. After that, we would have to re-generate all proofs for the proof obligation matrix to test if our new proposal was an inductive invariant. If it was not, then not all of the matrix's subgoals were preserved, and we had to repeat the above process. Since the number of conjuncts usually grew, the number of columns in the proof obligation matrix also grew. Discovering the correct conjuncts such that all proofs were accepted by Isabelle led to a final $70 * 800$ matrix! It was very clear from the middle of the proof that the proof would not be feasible without better tooling than that already existing in Isabelle/HOL. This inspired us to develop the **super_sketch** and **super_fix** tools.

Our tools, **super_sketch** and **super_fix**, attempt to minimise the human labour involved in the above task. They enabled us to prove correct properties of the cache-coherence protocol requiring dozens of iterations that ultimately led to an inductive invariant with nearly 800 conjuncts and over 54000 proof obligations. We built **super_sketch** upon Haftman's **sketch** tool [2], which automatically generates an Isar [3] skeleton for a single lemma in Isabelle/HOL. However, **super_sketch** extends this functionality by retrieving all proof goals emerging after the application of user-suggested proof methods and attempting to solve all of them concurrently. For more difficult subgoals, **super_sketch** tries additional user-supplied heuristics to reduce the subgoals before calling further automation on them. Application of **super_sketch** enabled us to rewrite the proofs of our rule lemmas in under 30 minutes each. To iterate through the entire obligation matrix thus required a whole working day. Fortunately, after regenerating the over 50k proof goals, **super_sketch** only failed to find a proof less than 100 times. To address those remaining failures, we implemented **super_fix**, which is a tool that simulates the action of a proof

engineer opening the failing file with the proof IDE (PIDE), locating the error and fixing it by replacing it with alternative text. In the end, a hybrid approach combining scripts 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. After a first application of **super_fix**, only 9 files remained with errors, 1 per file. A second call to **super_fix** completed the proof. These numbers under-approximate the total errors tackled with **super_fix** since some errors were not recorded in the tool's processing log.

Noticing the success of combining **super_sketch** and **super_fix** for automating proof workflows, we are firstly exploring how much of the proof process could be automated with a single command. For this, we have developed the **meta_sketch** tool that overcomes the manual scripting to handle the obligation matrix. Instead, **meta_sketch** provides a domain-specific language that enables users to supply a hierarchical sequence of proof methods that break goals into sub-goals, which we call *splitters*. After applying the user-sequence of splitters, the tool invokes a concurrent portfolio of strategies to try at the leaves of the proof tree and close all subgoals. Our leaf engine utilises a three-stage pipeline—racing, rescuing, and last-resort splitting—implemented via Isabelle/ML's Future constructs and ensures a "winner-takes-all" efficiency approach and a quick cancellation of losing branches. By allowing users to configure these strategies through our DSL of splitters, **meta_sketch** is eliminating the need for manual workarounds and encapsulates the entire proof-generation workflow into a single, extensible command.

Secondly, we have also integrated generative AI into the Isabelle proof assistant to explore and compare its sketching or **sorry**-removal abilities against our tools. Specifically, we have provided infrastructure for interacting with LLMs in Isabelle via a client-server architecture. That is, we have enabled Isabelle to call a Python-based server that can either host local Hugging Face [4] models, query other LLM-based servers such as Ollama [5], or use other APIs from state-of-the-art models such as ChatGPT [6] or Google's Gemini [7]. We have leveraged the DeepIsaHOL project's libraries [8] to build a collection of StandardML functions in charge of sending the required data to the LLM server. These library implementations result in two main commands for interacting with LLMs in Isabelle:

The command **llm_recommend** receives a line number pointing to the (line immediately after the) statement of a conjecture in a `.thy` file. As its name suggests, this function simply queries the LLM to suggest the next step in the proof. The command **llm_try_proof** iterates the same querying process to incrementally complete a proof string. Its implementation relies on a depth-first search (DFS) algorithm that returns when Isabelle confirms that a proof has been obtained or when a maximum number of proof (depth) steps is attained. At the moment, HF models can provide several suggestions, enabling the DFS to produce various candidate proofs. However, we only support one suggestion at a time for Ollama models. The proof depth is configured on the Isabelle side, while the breadth of the proof exploration comes from the model's (server) configuration file.

Thirdly, we have observed that most of the operations that we have defined while working on improving **super_sketch** and **super_fix** consist of functions that map the ITP's states to strings $f : S \to \Sigma^*$. Namely, we have identified that the interaction between a user and an ITP can be formalised as a sequence of state transitions $\sigma_i \to \sigma_{i+1}$ driven by strings $s_i \in \Sigma^*$, where a proof is successful if it reaches a state σ_m free of proof obligations and errors. Thus, we have explored the modular refactoring of our tools using these functions, which we refer to as *fixer functions*. For instance, our reimplementation of **super_sketch** using fixer functions, **try_sketch**, decomposes a goal into subgoals, generates preliminary "fixer strings" with **sorry** placeholders, and concurrently invokes user-supplied fixer functions to replace those placeholders with valid proofs. Similarly, **super_fix** leverages this model to sequentially traverse theory files, detect error states or **sorry** commands and apply fixer functions to repair them. We are exploring the benefits and gains of this re-implementation and have initially observed some efficiency gains when comparing **try_sketch** against **super_sketch**.

Finally, we have constantly evaluated our developments. For instance, we evaluated Isabelle through a large-scale stress test on an obligation matrix consisting of up to 54,128 subgoals (a 68×796 subset of the original one). The experimental data demonstrates that Isabelle effectively handles large-batch processing, achieving an average throughput of approximately 0.90s per goal. However, the evaluation revealed a significant performance trade-off: while concurrent execution is necessary to avoid the failures of sequential processing, excessive parallelism introduces thread contention, memory pressure, and scheduler delays. This contention frequently inflates the latency of trivial methods, triggering a "timeout cascade" that over-approximates failure rates—estimated between 16% and 40% using a **sorry**-based metric. Future work aims to stabilise these dynamics through adaptive timeouts and refined concurrency scheduling. We have made our tools publicly available [9].

Acknowledgements

This work was supported by EPSRC grant EP/R006865/1 and a Horizon MCSA 2022 Postdoctoral Fellowship (project number 101102608).

References

[1] C. Tan, A. F. Donaldson, J. J. Huerta y Munive, and J. Wickerson, "The burden of proof: Automated tooling for rapid iteration on large mechanised proofs," in *2025 IEEE/ACM 13th International Conference on Formal Methods in Software Engineering (FormaliSE)*, 2025, pp. 34–45, https://doi.org/10.1109/FormaliSE66629.2025.00010.

[2] 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

[3] 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

[4] Hugging Face, "Hugging face – the ai community building the future." online: https://huggingface.co/; accessed 20-October-2025. [Online]. Available: https://huggingface.co/

[5] Ollama, "Ollama – chat & build with open models." online: https://ollama.com/; accessed 20-October-2025. [Online]. Available: https://ollama.com/

[6] OpenAI, "Openai api platform," online: https://openai.com/api/; accessed 25-October-2025. [Online]. Available: https://chatgpt.com/

[7] Google, "Gemini api," online: https://ai.google.dev/gemini-api/docs; accessed 25-October-2025. [Online]. Available: https://gemini.google.com/app

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

[9] C. Tan and J. J. Huerta y Munive, "super_sketch_and_super_fix," Nov. 2025, https://github.com/ChengsongTan/super_sketch_and_super_fix.

I. Appendix: Figures

$$\begin{pmatrix} \ddots & & \iddots \\ & \begin{pmatrix} P\ \sigma \wedge \\ \mathtt{guard}_{R_i}\ \sigma \wedge \\ \sigma \rightarrow_{R_i} \sigma' \\ \Longrightarrow \\ \phi_j\ \sigma' \end{pmatrix}_{(i,j)} & \\ \iddots & & \ddots \end{pmatrix}_{m \times n}$$

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

```
lemma "(∑i≤(n::nat). 2*i-1) = n^2"
(*start state σ_φ after initial string s_0*)

try_sketch SORRYS[simp] (induct n)
proof(induct n)

  show goal0:"(∑i::nat≤0. 2*i-1)=0^2"
    by simp
next
  show goal1:"(∑i≤Suc n. 2*i-1)=(Suc n)^2"
    if "(∑i≤n. 2*i- 1) = n^2"
    for n :: "nat"
    using that
    sorry
qed
(* intended final substring s_m turned into
σ_m once the induct step is solved *)
```

Fig. 2: Example of a proof produced by **try_sketch**. Green/blue highlights mark `simp` and `induct n`. Red/orange/purple dots and same-colored labels align with the exact end of the preceding code lines, indicating when the proof state reaches σ_φ, σ_k, and σ_m.

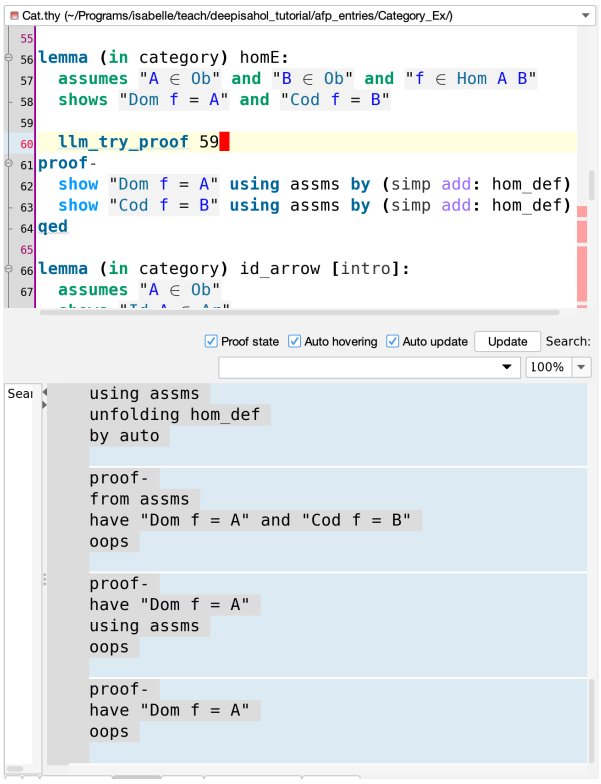


Fig. 3: Example of the command **llm_try_proof** interacting with a local LLM to produce a proof for the theorem `homE` in the AFP entry on Category Theory.

```
1 super_fix HAMMER[simp] "path/to/file"
```

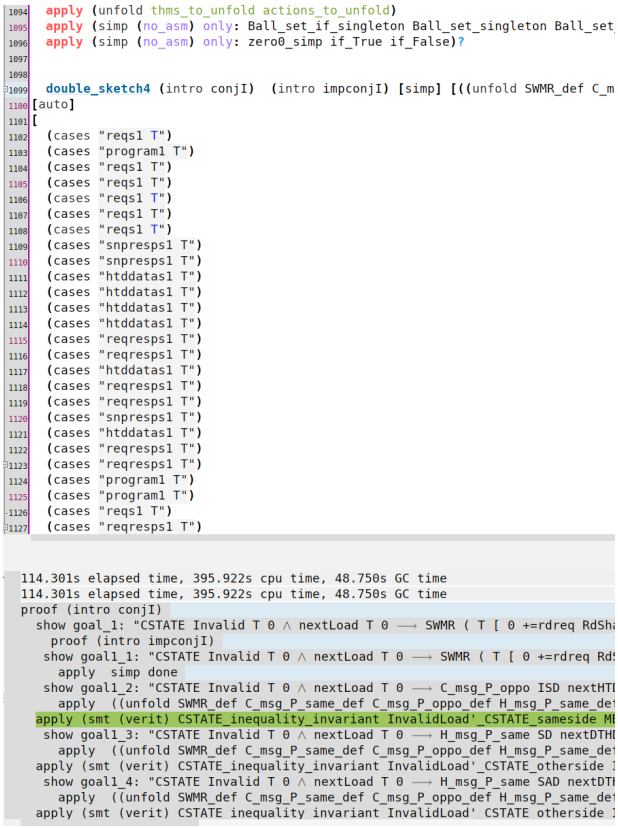
Super_Fix_Demo.thy (~/Programs/isabelle/teach/deepisahol_tutorial/s...

```
1  theory Super_Fix_Demo
2    imports Complex_Main
3
4  begin
5
6
7  lemma real_sqrt_le_iff': "x ≥ 0 ⟹ y ≥ 0
8    ⟹ sqrt x ≤ y ⟷ x ≤ y ^ 2"
9    apply blast
10   done
11
12 lemma sqrt_le_itself: "1 ≤ x ⟹ sqrt x ≤ x"
13   sorry
14
15 lemma sqrt_real_nat_le:"sqrt (real n) ≤ real n"
16   apply simp
17   done
18
19 lemma semiring_factor_left:
20   fixes b:: "'a::semiring"
21   shows "a * b + a * c = a * (b + c)"
22   apply simp
23   done
24
25 lemma sin_cos_squared_add3:
26   fixes x:: "'a:: {banach,real_normed_field}"
27   shows "x * (sin t)² + x * (cos t)² = x"
28   apply simp
29   done
30
31 lemma sin_cos_squared_add4:
32   fixes x:: "'a:: {banach,real_normed_field}"
33   shows "x * (cos t)² + x * (sin t)² = x"
34   apply simp
35   done
36
37 lemma "φ ⟶ (ψ ⟶ φ)"
38   sorry
39
40 lemma "(φ ∨ φ) = (φ ∧ φ)"
41   sorry
42
43 lemma "¦¦a¦ - ¦b¦¦ ≤ ¦a - b¦" for a::real
44   apply rule
45   done
```

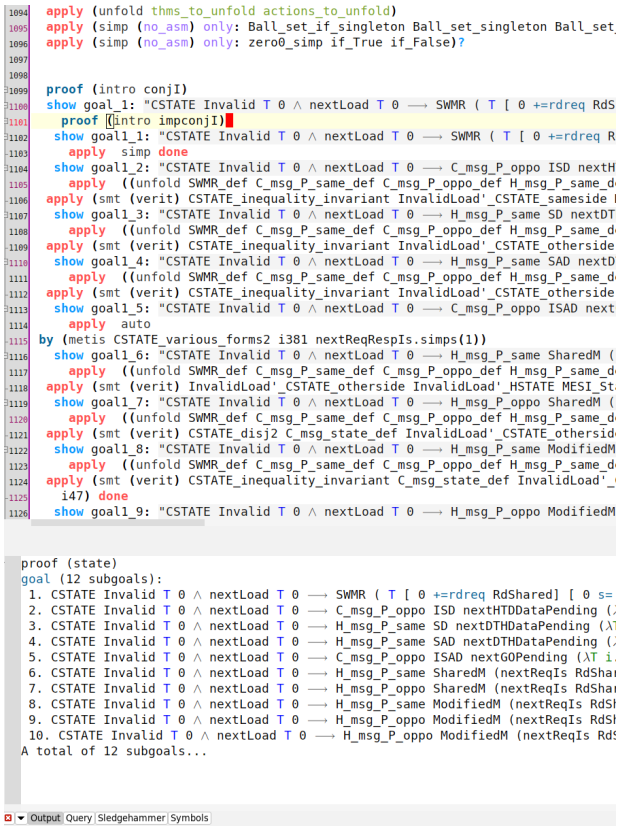
Super_Fix_Demo_Fixed.thy (~/Programs/isabelle/teach/deepisahol_tu...

```
1  theory Super_Fix_Demo_Fixed
2    imports Complex_Main
3
4  begin
5
6
7  lemma real_sqrt_le_iff': "x ≥ 0 ⟹ y ≥ 0
8    ⟹ sqrt x ≤ y ⟷ x ≤ y ^ 2"
9    by (smt (verit, ccfv_SIG) real_le_lsqrt sqrt_le
10
11
12 lemma sqrt_le_itself: "1 ≤ x ⟹ sqrt x ≤ x"
13   sorry
14
15 lemma sqrt_real_nat_le:"sqrt (real n) ≤ real n"
16   by (smt (verit, ccfv_SIG) less_Suc0 nat_1 nat_
17
18
19 lemma semiring_factor_left:
20   fixes b:: "'a::semiring"
21   shows "a * b + a * c = a * (b + c)"
22   by (smt (verit, del_insts) distrib_left)
23
24
25 lemma sin_cos_squared_add3:
26   fixes x:: "'a:: {banach,real_normed_field}"
27   shows "x * (sin t)² + x * (cos t)² = x"
28   by (smt (verit, del_insts) One_nat_def cos_of_
29
30
31 lemma sin_cos_squared_add4:
32   fixes x:: "'a:: {banach,real_normed_field}"
33   shows "x * (cos t)² + x * (sin t)² = x"
34   by (simp add: semiring_factor_left)
35
36 lemma "φ ⟶ (ψ ⟶ φ)"
37   by (smt (verit, del_insts))
38
39 lemma "(φ ∨ φ) = (φ ∧ φ)"
40   by (smt (verit, ccfv_threshold))
41
42 lemma "¦¦a¦ - ¦b¦¦ ≤ ¦a - b¦" for a::real
43   by (smt (verit))
44
45
```

Fig. 4: Example of a file fixed with **super_fix**. We show (1) at the top, the template command used to call the tool, (2) on the left, the file with errors, and (3) on the right, the fixed file.



(a) **double_sketch** finished, with time information and proof suggestion



(b) **double_sketch** suggestion got accepted

Fig. 5: Side-by-side comparison of **double_sketch** command suggestion being adopted.