RAMTERA TECHNICAL WHITE PAPER SERIES

WP-06

Landing in the Incumbent's Debug Flow

Results database interoperability for physical verification tools

A verification tool that cannot put its markers where the engineer already looks has not finished the job.

ASCII RDB	SKILL	SQLite	3
RESULTS FORMAT	LAYOUT OVERLAY	QUERYABLE STORE	FORMAT TRAPS DOCUMENTED

Document	WP-06 — Landing in the Incumbent's Debug Flow
Author	Sameer Pawanekar, Founder & Chief Executive Officer, Ramtera Inc.
Revision	v1.0
Date	August 2026
Classification	Public

ABSTRACT

A physical verification engine produces violations. An engineer fixes them, and does so inside a layout editor, using a results browser, in a flow built around whichever tool the organisation already owns. A challenger tool that emits results in its own format asks every user to change how they work before they have any reason to trust it, which is the wrong order. This paper describes what interoperability actually requires: writing an established results format correctly — including three specific traps in it that are not obvious from inspection — pushing markers into the layout editor as native objects, and holding results in a queryable store rather than a flat file.

01 The adoption argument

The barrier to adopting a new verification tool is rarely the engine. It is that the engine's output arrives somewhere nobody looks, in a form nothing else reads.

A design organisation has an established debug loop: run verification, open the results browser, cross-probe to the layout editor, fix the geometry, re-run. That loop is built around a specific results format and a specific editor integration. A tool whose output sits outside it is not evaluated on its merits, because evaluating it requires the engineer to first abandon a workflow that works.

> **BE INVISIBLE IN THE FLOW**
> The correct posture for a challenger is to be invisible in the flow. The engineer should be able to run the new engine, open the same browser, cross-probe to the same editor, and only notice the difference in the runtime and the results themselves. Interoperability is not a feature that comes after the engine is good; it is what makes it possible for anyone to find out whether the engine is good.

02 Writing an established results format correctly

Results database formats used in physical verification are simple in structure — a rule header, a marker count, coordinate records — and are usually undocumented in their finer points. Writing one that a third-party browser accepts requires getting details right that are not inferable from an example file. Three cost us time and are worth naming.

Trap one: the count field means something specific

Marker records carry a count field whose interpretation is not the obvious one. Writing the number of coordinate pairs, or the number of vertices, or the number of markers, produces a file that parses without complaint and displays incorrectly — typically with markers missing or merged. The field has a precise meaning tied to the record structure, and the failure mode when it is wrong is silent rather than diagnostic.

Trap two: rule names cannot be empty

A rule that produces results but has no name — which can arise from a derived or internal check — will emit a record with an empty name field. Browsers handle this inconsistently: some display nothing, some display the previous rule's results under the wrong heading, and some fail to load the file entirely. Every emitted rule needs a name, synthesised if necessary.

Trap three: tiled execution duplicates results at boundaries

An engine that partitions the design into tiles for parallel execution will find the same violation from more than one tile where a marker spans a boundary. Written naively, the results file contains duplicates, which inflates every count in it and corrupts any correlation study performed against it.

The remedy is ownership: each marker is attributed to exactly one tile by a deterministic rule — for instance, the tile containing a designated corner of the marker — and emitted only by that tile. This must be deterministic rather than first-writer-wins, or the output varies between runs at the same tile count.

WHY THESE ARE HARD TO CATCH

All three traps share a property: the file remains syntactically valid and the failure is visual or numerical rather than structural. There is no parse error to catch. The only reliable test is to open the output in the browser it is intended for and compare against a known-good file from the incumbent tool.

03 Markers as native objects in the layout editor

A results browser tells the engineer that a violation exists. Fixing it requires seeing it in place, on the geometry, in the editor. The mechanism that matters here is pushing markers into the editor as native marker objects rather than as an imported layer of shapes.

Approach	What the engineer gets	Limitation
Import markers as a layout layer	Shapes visible in the canvas	Not selectable as violations; no cross-probe; pollutes the design database
Create native marker objects	Selectable, cross-probeable violations with rule identity attached	Requires editor scripting integration
Screenshot or report only	A list of coordinates	The engineer navigates manually

Table 1 — Approaches to putting violations in front of the engineer.

Native markers are worth the integration effort because they carry identity. A marker that knows which rule produced it can be filtered, dismissed, waived, and cross-referenced. A shape on a layer cannot.

Round-tripping the selection

The integration is more valuable if it works in both directions. An engineer who selects a marker in the editor and marks it as reviewed or waived is generating information that belongs back in the results database. Pushing markers out is table stakes; reading the engineer's disposition back is what turns the integration into part of the workflow rather than a display feature.

04 A queryable results store

Verification results are conventionally written as flat files. At small scale this is fine. At the scale a full-chip run produces, a flat file is the wrong data structure for every question anyone actually asks of it.

Question the engineer asks	Flat file	Queryable store
Which rules have violations, ranked by count?	Full scan and aggregate	One query
Show me violations in this region only	Full scan	Indexed lookup
What changed between this run and the last?	Two full scans and a diff	Join across runs
Which of these were already waived?	Not answerable within the file	Join against waivers
Which rules regressed since the last release?	Manual comparison	One query across runs

Table 2 — Questions a results store answers that a flat file does not.

An embedded relational store adds no operational burden — there is no server, no configuration, and the database is a single file that can be copied and archived like any other artefact. What it adds is that results become queryable, comparable across runs, and joinable against other data such as waiver records.

> **THE CAPABILITY THAT JUSTIFIES THE STORE**
> The run-to-run comparison capability is the one that changes behaviour. Once results from successive runs are in the same store, “what did my last change break?” becomes a query rather than a project. That question is asked constantly and answered badly almost everywhere.

05 What interoperability is not

It is not a substitute for the engine being correct

Excellent integration around a tool that produces wrong results makes the wrong results easier to look at. Interoperability lowers the barrier to evaluation; it does not affect the outcome of the evaluation.

It is not an argument for adopting the incumbent's data model wholesale

Writing a format correctly is a compatibility decision, not an architectural one. The internal representation should be whatever serves the engine; the established format is an export target. Conflating the two constrains the engine for no benefit.

It is not finished when the file parses

As all three traps above illustrate, a file that loads without error can still be wrong. The acceptance criterion is that the output displays identically to a known-good reference file from the incumbent tool for the same design and the same rules — compared marker by marker, not by opening both and looking at them.

06 Conclusion

Interoperability work is unglamorous and disproportionately determines whether anyone ever evaluates the engine behind it. The three components that matter are an established results format written correctly down to its undocumented details, markers delivered into the layout editor as native objects carrying rule identity, and a results store that supports the queries engineers actually run — particularly comparison across runs.

None of this is difficult. All of it is easy to defer, and deferring it means the engine is judged on how awkward it is to use rather than on what it finds.

About Ramtera

Ramtera Inc. develops **LVR**, a physical verification engine that executes foundry SVRF rule decks natively, covering design rule checking, layout versus schematic, electrical rule checking, layout XOR, density and fill, and parasitic extraction. LVR is written in modern C++ with a Qt6 interface, and has been validated against publicly available process design kits including SKY130 and IHP SG13G2.

Ramtera also develops **PRE**, a place-and-route engine sharing LVR's geometry kernel, and maintains a portfolio of granted patents, registered copyrights and trademarks across its verification and implementation technologies.

Ramtera Inc. is incorporated in the United States, with research and development conducted by Ramtera Design Automation India Private Limited in Bangalore, India.

Contact	
Web	https://ramtera.com
General enquiries	info@ramtera.com
Technical support	support@ramtera.com
Series	Ramtera Technical White Paper Series
This document	WP-06, revision v1.0, August 2026