RAMTERA TECHNICAL WHITE PAPER SERIES

WP-07

The Netlist Parser Nobody Writes About

Robustness in SPICE and CDL ingestion for layout versus schematic

Most reported LVS mismatches that turn out not to be mismatches are parser defects wearing a circuit problem's clothing.

4	0	100%	1
FAILURE CLASSES	OF THEM CIRCUIT BUGS	SILENTLY MISDIAGNOSED	CORRECT FIX EACH

Document	WP-07 — The Netlist Parser Nobody Writes About
Author	Sameer Pawanekar, Founder & Chief Executive Officer, Ramtera Inc.
Revision	v1.0
Date	August 2026
Classification	Public

ABSTRACT

Layout versus schematic compares two netlists. One of them is parsed from a text file whose format is defined by convention, extended by every simulator that has ever read it, and produced by tools that each interpret the conventions slightly differently. A defect in that parser does not present as a parse error. It presents as a device count mismatch, a missing net, or a parameter disagreement — that is, as a circuit problem — and sends the engineer looking in entirely the wrong place. This paper documents four such failure classes, why each misdiagnoses, and what the correct handling is.

01 Why parser defects misdiagnose

A netlist parser that fails loudly is a solved problem. The parser failures that cost time are the ones that succeed: they produce a netlist, it is well-formed, and it is not the netlist in the file.

Downstream, the comparison engine does exactly what it should — it reports that the two netlists differ. The engineer receives a device count mismatch and begins investigating the layout, because that is what a device count mismatch means. The investigation is thorough, competent and aimed at the wrong artefact.

> **THE CHECK TO RUN BEFORE INVESTIGATING ANY MISMATCH**
> The diagnostic that cuts through this is cheap and underused: before investigating any reported mismatch, dump the parsed netlist and compare its device and net counts against the source file counted independently. If they disagree, the problem is upstream of the comparison and no amount of layout analysis will find it.

02 Failure class one: continuation lines

A netlist statement can span multiple physical lines, with continuation indicated by a leading marker character. A parser that processes the file line by line without reassembling continuations will read a continued device statement as a complete statement followed by an unrecognised line.

How it presents

The truncated statement often remains syntactically plausible — a device with fewer terminals or fewer parameters than intended — so it is accepted. The continuation line is skipped as unrecognised. The result is a device with wrong parameters and a silently discarded fragment.

Correct handling

Reassemble the logical statement before tokenising anything. This must happen at the lowest level of the reader, not as a special case inside statement handlers, or every handler needs to know about it and one of them will not.

03 Failure class two: fixed-size buffers

Netlist readers commonly use fixed-size character buffers for line handling. Hierarchical instance names in a real design are long — deep hierarchy, bus notation and generated names combine to produce identifiers that exceed any buffer size chosen on the assumption that names are short.

How it presents

The optimistic outcome is a crash, which is at least diagnostic. The problematic outcome is a truncated name, which parses successfully and produces a distinct net or instance that happens to share a prefix with another. Two nets that should be one, or two instances that collide, both produce comparison failures that look like connectivity errors in the layout.

TRUNCATION IS THE DANGEROUS CASE

Truncation is worse than overflow. An overflow is caught by a sanitiser or a crash; a truncation produces a valid identifier that is quietly the wrong identifier. Any fixed-size buffer in a netlist path should either be replaced with a growable string or made to fail explicitly on overrun.

Correct handling

Use growable strings throughout. Where a fixed buffer is retained for performance in a hot path, it must detect overrun and fail explicitly rather than truncating.

04 Failure class three: ports versus instances

Within a subcircuit definition, a line listing identifiers can be a port declaration or an instance line depending on where it appears relative to the definition header. The distinction is positional rather than syntactic, and a parser that determines it by inspecting the line content alone will sometimes get it wrong.

How it presents

A port list read as an instance produces a spurious device. An instance read as a port list produces a missing device and a set of nets that appear in the interface but connect to nothing. Both present as device count mismatches.

Correct handling

Track position within the subcircuit definition explicitly, so that the classification is determined by parser state rather than inferred from the line. Content-based heuristics work on the common case and fail on precisely the designs where a mistake is most expensive to find.

05 Failure class four: the multiplier attribute

A netlist expresses repeated devices using a multiplier attribute rather than instantiating each device separately. A device with a multiplier of eight is eight devices in the circuit and one line in the file. The layout, meanwhile, contains eight structures.

How it presents

A parser that treats the multiplied device as one device reports a device count mismatch against a layout that is entirely correct. The magnitude of the mismatch is a function of how many multiplied devices the design contains, which makes it look design-specific and therefore like a real circuit issue.

Correct handling

Expand at parse time, and then reconcile the expanded count against the structure references present in the layout database. Expansion without reconciliation converts an obvious mismatch into a subtle one, because the counts now agree in the common case and disagree only where the layout uses a different repetition mechanism than expected.

GENERALISE RATHER THAN SPECIAL-CASE

The multiplier attribute is not the only key-value parameter that affects device identity, and a parser that special-cases it will meet the same problem again with the next one. Handle key-value parameters generally, and treat multiplication as one consequence of a parameter rather than as a syntactic feature.

06 Summary of the four classes

Failure class	Presents as	Correct handling
Continuation lines not reassembled	Wrong parameters; silently discarded fragments	Reassemble logical statements before tokenising
Fixed-size buffer truncation	Colliding or split nets and instances; connectivity errors	Growable strings; explicit failure on overrun
Port list misclassified as instance	Spurious or missing devices	Positional classification from parser state
Multiplier attribute not expanded	Device count mismatch on a correct layout	Expand at parse time; reconcile against layout structure references

Table 1 — The four failure classes, their presentation, and their remedies.

THE COMMON THREAD

Every one of these presents as a circuit problem and none of them is one. That is the common thread, and it is the reason a netlist parser deserves the same testing rigour as the comparison engine downstream of it — which it almost never receives, because parsing is assumed to be solved.

07 Testing a netlist parser

1. **Round-trip the netlist.** Parse it, write it back out, parse the result, and compare the two parsed forms. This catches a large fraction of ingestion defects without needing a reference tool.
2. **Count independently.** Devices and nets counted from the source file by a simple script should match the parsed model. A disagreement localises the fault upstream of the comparison immediately.
3. **Test with adversarial names.** Long hierarchical identifiers, bus notation, and generated names belong in the test corpus deliberately, because they are what real designs contain and what synthetic test files never do.
4. **Test continuation at every statement type.** Continuation handling that works for device lines and not for subcircuit headers will pass a naive test suite and fail on real input.
5. **Make the parsed netlist dumpable.** The ability to emit the model as text is the single most useful debugging feature a netlist reader can have, and it costs almost nothing to build.

08 Conclusion

Netlist parsing attracts no attention because it appears to be a solved problem, and it appears to be solved because its failures are attributed elsewhere. Every failure class described here produces a well-formed netlist that differs from the file, and therefore a comparison result that correctly reports a difference the engineer then investigates in the wrong artefact.

The disproportionately valuable habit is to verify the parsed model against the source before investigating any reported mismatch. It takes a minute and eliminates an entire category of misdirected debugging.

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-07, revision v1.0, August 2026