VCL2FMXConverter v5 Contract System Overview

How the converter, contracts, expectations, and real-world test data work together

Purpose

Version 5.0 Vanguard - Revision date: June 27, 2026

The contract system is a safety net around the converter. It does not learn from project data or change rules dynamically. Instead, each contract is a small known problem case with an expected result.

[v5.0 current contract baseline] The current v5.0 suite contains 198 executable expectations. The suite includes dry-run preview accounting, report-card layout, include analysis, Windows messaging categories, DFM/FMXL fidelity checks, semantic structure checks, mapping assistance, graphics/GDI handling, and protected uses cleanup.

The converter is run against that input, and the contract runner checks whether the generated output, uses clauses, FMX files, and report items match what we said should happen.

How It Fits Together

The converter still does the real work. It reads a VCL project or unit, parses Pascal and DFM content, applies conversion rules, writes FMX output, and produces a conversion report.

The contracts sit outside that process as verification. They prove that known conversion rules continue to behave correctly after later changes.

The Converter Does This

1. Reads a VCL project or unit.
2. Parses Pascal, DFM, project files, include files, forms, uses clauses, and known risky patterns.
3. Rewrites VCL units, components, properties, and events into FMX equivalents where it safely can.
4. Copies or preserves things that should not be rewritten automatically.
5. Creates generated FMX Pascal, DFM, and FMX output.
6. Creates a conversion report showing what was converted, what needs review, and what may block reliable output.

The Contracts Verify This

Each contract says, in effect: Given this input file, the converter must produce this kind of output, must not produce that dangerous output, must add these units, must remove those units, and must report these issues.

The contracts are not part of the user's converted application. They are test fixtures that prove the converter behaves correctly.

What A Contract Contains

A typical contract has two parts.

Sample Input File

The sample input file contains the problem pattern being tested. For example:

```
contracts\components_and_events\19_form_centering_float_rounding.pas
```

Expectation File

The matching expectation file describes what must happen after conversion. For example:

```
contracts\components_and_events\19_form_centering_float_rounding.expected.json
```

The .pas file contains the source pattern. The .expected.json file contains the rules that the generated output and report must satisfy.

Example Contract Rule

The new form-centering contract came from a real FMX compile problem. The unsafe converted output looked like this:

```
AForm.left := (ScreenWidth - AForm.Width) / 2;
```

That fails because FMX Width is floating-point and Left is integer. The safe converted output should be:

```
AForm.Left := Round((ScreenWidth - AForm.Width) / 2);
```

The contract therefore checks both sides of the rule.

- The converted output must contain the rounded FMX-safe assignment.
- The converted output must not contain the unsafe unrounded assignment.

What The Contract Runner Does

When the contract suite runs, the runner performs the same conversion process against each fixture, then compares the result against the expectation file.

7. Finds every contract fixture.
8. Copies the fixture into a temporary test output area.
9. Runs the converter against that fixture.
10. Reads the generated Pascal, FMX, and report output.
11. Compares the output against the expectation JSON file.
12. Fails if anything required is missing.
13. Fails if anything forbidden appears.
14. Fails if a fixture has no expectation when coverage enforcement is enabled.

What The Contracts Do With Presented Data

If the presented data is a contract fixture, the contracts validate it directly. They check the generated output, generated uses clauses, generated FMX files, and generated report text.

A contract can check questions such as these:

- Did Vcl.Forms disappear?
- Did FMX.Forms appear?
- Did Winapi.Messages get removed when no longer needed?
- Did a false positive avoid producing a report?
- Did a real Windows message produce a manual-review report?
- Did GDI drawing become FMX canvas-style code?
- Did include files get copied and analyzed?
- Did unsafe output get blocked?
- Did generated code avoid known Delphi compile errors?

If the presented data is a real project, like Carillon, the contracts do not directly act on it. The converter acts on the real project.

When a real-world failure is found, we make a small generalized contract that represents that class of failure. That way the fix protects all users, not just one project.

Current Contract Families

The v5 contracts now cover several important conversion areas.

- Include-file analysis, including missing, nested, recursive, outside-tree, conditional, and UTF-8 includes.
- Windows messaging, including SendMessage, PostMessage, Perform, WndProc, message declarations, WM_USER, system commands, and false positives.
- Uses-clause cleanup, including VCL and Winapi units, conditional protected uses blocks, and leftover Vcl.Themes.
- GDI and graphics conversion, including FMX canvas output and visual-review reporting.
- DFM and FMX generation, including string collections, TStringGrid event order, and UTF-8 output.
- Components and events, including form lifecycle, timers, media, image properties, and coordinate conversion.
- Encoding, reporting, data-aware conversion, LiveBindings, project files, and third-party mapping behavior.

Why This Matters

Before v5, fixes could be made by intuition. That worked for some cases, but each fix risked breaking another case.

With contracts, each known problem becomes a permanent rule. The converter can keep improving while the contracts guard against regressions.

That gives v5 a much stronger foundation. Every structural converter change has proof behind it.

Practical Workflow

The working loop is simple.

15. A real project exposes a converter failure or suspicious output.
16. We reduce that failure to a small generalized contract fixture.
17. We write expectations for the safe output and forbidden output.
18. We run the contract suite and confirm the failing behavior is captured.
19. We fix the converter.
20. We rerun contracts and regression guards.
21. The fixed behavior becomes part of the permanent v5 safety net.

In short, the contracts turn real-world conversion problems into repeatable proof. They do not make the converter project-specific. They make the converter safer for everyone.