Compilation of feature models by expert vibing

Vadim Zaytsev

Formal Methods and Tools, University of Twente, Enschede, The Netherlands

Abstract

This paper documents the author's participation in the 2025 Transformation Tool Contest (TTC) Live Contest, which focused on the automatic translation of feature models specified in the Universal Variability Language (UVL) into the Graphviz DOT format. The challenge was approached through the development of a minimalist domain-specific language workbench in F#, reflecting the combined influence of software language engineering expertise and an emerging software development methodology called *"expert vibing"* — the judicious and unapologetic use of support that generative artificial intelligence can provide. The resulting solution demonstrates that concise and performant model transformations can be achieved outside conventional frameworks by leveraging both custom lightweight languages and AI-assisted coding practices. We present Indentia and Scripta languages, discuss practical and theoretical trade-offs compared to established tools, and empirically evaluate correctness, conciseness, and performance against the reference solution. The findings highlight both the potential and current limitations of large language models for model transformation tasks, and provide insight into the evolving workflow of *vibe coding* in software engineering.

Keywords

Live contest, language workbench, indentation-based parsing, DSL design

1. Introduction

The *Transformation Tool Contest* (TTC) [1] is a long-running series of competitions, first emerging from predecessor events such as AGTIVE (2007) and GraBaTs (2008), and officially continuing near-yearly as TTC since 2010. It aims to compare the expressiveness, usability and performance of transformation tools across a variety of challenging model-to-model and model-to-text case studies.

This report documents its author's participation in the *2025 TTC Live Contest* at STAF 2025 [2], which was held in the trend of its predecessors: Java annotation processing in 2015 [3] and incremental graph queries in 2018 [4]. In 2025, the real goal of the contest was to see how far Large Language Models (LLMs) are at that moment in generating code for model transformation problems, and the case study was chosen to be about transforming feature models specified in UVL [5] into Graphviz DOT format [6]. The goal of this report is twofold: first, to reflect on the process of delivering a working transformation with an LLM help under time pressure and other constraints imposed by the organisers; and second, to present a concise documentation of the final deliverable, which is a minimalistic and performance-conscious implementation. The expectation is that this solution is sufficiently different from alternatives, and the path towards it is relatively full of circumstances worth discussing.

The remainder of this report is organised as follows: in section 2 we summarise the live contest case of this year, to keep the paper self-contained. In section 3 we explain the basic principles behind the envisioned solution, and some design decisions that shaped it. The section 4 presents two software languages that formed the core of the solution. The section 5 describes phases of the benchmarking framework, and section 6 reports on the collected measurements and compares observations of the reference solution against this newly proposed one. The paper concludes with section 7.

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10–13, 2025

vadim@grammarware.net (V. Zaytsev)
https://grammarware.github.io (V. Zaytsev)
0000-0001-7764-4224 (V. Zaytsev)

2. The Problem: Case Description

The 2025 TTC Live Contest centres on the problem of translating *feature models* written in the Universal Variability Language (UVL) [5] to Graphviz's DOT format [6]. This case was motivated by the current advances in large language models (LLMs) and their rapidly improving capabilities in code generation. The contest organisers specifically sought to explore to what extent generative AI tools are able to tackle model transformation problems that have traditionally been addressed with specialised model transformation languages and frameworks.

The fundamental **research question** of the contest was: *How capable are large language models (LLMs) at generating correct, efficient, and maintainable model transformations, compared to traditional manual approaches and specialised transformation tools?* To this end, the contest case was intentionally chosen to be approachable for both humans and AI-producing complex XMI representations in favour of textual domain-specific languages as both source and target.

Participants were tasked with designing an automated transformation from UVL — a human-readable, indentation-based DSL for feature modelling — to DOT, the widely-used textual graph description language underlying Graphviz. The goal was to produce a DOT representation faithful to the semantics of the input feature model, including its hierarchy, feature groups (mandatory, optional, or, alternative), and constraints.

The specific challenges included:

- Parsing UVL, which requires understanding indentation to recover group structure.
- Mapping semantic elements (features, groups, constraints) to the corresponding DOT constructs and properties.
- Reproducing reference rendering and attribute conventions, such as node colours, shapes, edge styles, and inclusion of feature model constraints as HTML tables in DOT.
- Optionally, supporting incremental change propagation based on provided model diffs or at least the model files in each series.

A public benchmark suite was provided, supporting automation of execution and profiling, as well as a variety of existing UVL models to try. Solutions were to be evaluated on criteria such as *correctness*, *understandability*, *conciseness*, and *execution time*. Notably, participants were asked to document their use of generative AI (if any), including the nature of LLM prompts and required manual adjustments.

This live contest, therefore, is not only meant to test technical skills in software language engineering and transformation, but also contributed to the ongoing empirical study of the role of LLMs in model-driven engineering.

3. The Solution: Software Language Engineering

Software language engineering (e.g., [7]) is a field of computer science and software engineering, focused on collecting techniques from various domains and reusing them across technological spaces. These techniques are based around a concept of a *software language*, which is an umbrella term covering general purpose programming languages and domain-specific languages, but also phenomena such as markup languages which can differ quite radically on the very concepts of otherwise basic activities like "correctness checking" or "execution". Software language engineering is a natural extension and generalisation of compiler construction, which absorbed and internalised much knowledge from software modelling, XML processing, DSL engineering, metaprogramming, etc.

One of the central ideas of software language engineering is a concept of a *language workbench* [8], as a (meta)tool that allows the software language engineer to leverage software language engineering techniques themselves and use domain-specific languages to develop domain-specific languages. That way, techniques like language-oriented programming [9], which treat languages as first-class artefacts at development time, stop from being a dream and move towards a daily routine: we can focus our designs and solutions around a collection of languages, and let generative techniques [10] fill in the technical gaps.

In the solution presented in this paper, we embrace this approach to the extreme. Rather than using existing transformation frameworks or general-purpose languages in a conventional way, we opted to build a lightweight, domain-specific language workbench designed exclusively for this contest. (Which is both a "*Domain-Specific Language* Workbench" in the sense that it serves as a workbench to make DSLs, and a "Domain-Specific *Language Workbench*" in the sense that this is a language workbench which is specific to a domain). This allowed both the involved languages and the transformation logic itself to be specified in a highly concise, declarative form, eliminating or minimising boilerplate and focusing on the essence of the mapping.

The final deliverable is essentially a one-off workbench. Do not try this at home. Or rather, do, but with caution and without overly high expectations. This solution is produced by combining:

- already available expertise in topics like parsing, grammar engineering, program transformation, functional programming and compiler construction;
- AI-assisted vibing through implementation details, as advised in the case description [2]; and
- aggressive cutting of all possible corners on the way to the final goal, specifically focusing on conciseness as one of the benchmarks.

Instead of relying on an external language workbench, we developed a minimal infrastructure in F# which ticks all or most of the boxes: it parses a "metamodel" and a transformation specification, loads the model, parses it according to the metamodel, transforms it and produces a compilable result for Graphviz.

3.1. Expert Vibing

As AI support, we have relied on ChatGPT 4.1. As the title of the paper suggests, the process was a bit different than the usual envisioned *vibe coding* [11], "where you fully give in to the vibes, embrace exponentials, and forget that the code even exists". Instead, this emergent methodology could be broken down into the following steps:

- give in to the vibes, use human expertise to recognise patterns in the problem that can be leveraged;
- contemplate possible solutions until a point of "this could work!" is reached;
- formulate a prompt explaining the gist of the envisioned solution;
- copy, paste and run the code;
- if any build errors or syntax errors manifest, complain back to the LLM until it finds a way to fix them;
- gain some evidence that the code actually works to some extent (e.g., by inspecting the results of a single run);
- move on to the next phase and iterate;
- use human software engineer expertise to detect code that smells like either being too lengthy or too slow, refactor it (usually within the IDE, or asking AI for anything that takes more than a few clicks).

The last point was essential to get to the presented results, but exhibited diminishing returns: any attempts to profile the code to some level of detailed comprehension at the end of the project were taking too much development time and effort and eventually ceased to produce any tangible measurable impact. It did help to produce a performance-competitive solution in merely 150 lines of F# code.

While the LLM offered useful suggestions, particularly for parser structure and boilerplate, much of the resulting code had to be reworked to align with the minimalist and language-oriented style. In many cases, this process exposed the limitations of current LLMs for compiler tasks that demand custom parsing or advanced DSL use.

3.2. Tool and Language Selection

Embracing the spirit of vibe coding at this live contest, the first prompt to ChatGPT 4.1 was about the choice of a language to implement it with, given that we will be parsing indentation-based language (UVL) and going for the shortest solution. ChatGPT advised Python, with the first argument being:

> *Naturally suited for indentation-based parsing, as Python itself uses indentation for code blocks.*

This is a blatant hallucination, which confabulates a link between a property of a programming language and the ease to implement an algorithm of dealing with a conceptually related property of a different language. There might be some truth to the Conway's Law [12], but it is a sociological phenomenon that does not manifest on a small scale in tangible technical consequences.

ChatGPT also added a more reasonable argument that Python has libraries like PyParsing [13] and Lark [14], but it was too late in the thinking process to convince anyone that using an existing parsing framework would lead to the best/shortest/fastest solution. Even in the industry, with all the tight deadlines and the lack of academic focus, it has happened to this very author at least twice (for *Engage!* [15] and PAX [16]) that he needed to implement a bespoke metatool (workbench/generator) to solve only one problem with it simply because real-life problems are often not efficiently solvable with commoditised off-the-shelf solutions.

The other two suggestions were Haskell and OCaml, which are both reasonable options for the reasons of code conciseness as well as for matching the available human expertise of the author, and not for the — also provided by ChatGPT — reason of relying on Parsec [17] or Menhir [18]. Taking this into consideration, we made a counter-proposal of F# as something in between. ChatGPT fully agreed, also supporting our argument that the author's familiarity with Roslyn [19], while not being a workbench nor a generator, could come in handy later in the project if some stronger backend support would end up being required. In the end, it ended up not being the case, and the solution as it was delivered, did not use any F# feature that would not be possible to port to a similarly powerful language.

The conciseness was quite satisfactory, even though F# enforces complete pattern matches which means some lines of code are lost to `| _ -> ()` in `match` statements. As a form of soft validation of this, once the project was completed, we have asked ChatGPT o3 (which is better at straight zero-shot one-prompt coding requests) to translate the core code from F# to Haskell (resulting in **323** LOC) and to OCaml (**344** LOC). The lines of code were counted including empty lines that separate function definitions or code blocks, but after removing all full line comments — in the same way we counted the lines in our own F# solution. We see this doubling of LOC as a soft confirmation of the correctness of our choice of language — either translation can be optimised further, so a hard confirmation is out of scope for us.

4. Indentia and Scripta

Before we dive into the details of how the solution works, let us first discuss the two new languages which formed the core of it. These are called Indentia for parsing indentation-based hierarchical data structures and Scripta for declarative model to text transformations of deep but uniformly structured data. Parsing can take many forms [20], and it is important to spot the aspects of this TTC case that make this solution work.

UVL is an indentation-based language, which means that technically we do not even need a grammar as a parsing specification. It can be handled with a simple generic algorithm that goes through the input line by line, creates a new inner container each time the indentation increases, and pops it back off each time the indentation decreases. However, UVL is a slightly stricter than that, since it observes five keywords: `features`, `mandatory`, `optional`, `alternative` and `or`, and we can accommodate for them directly in the grammar. Technically the metamodel is then in the combination of the grammar with the type which we ended up defining directly in F#:

```
type Feature = { Name: string;                      Constraints: ResizeArray<string>
                 Mandatory: ResizeArray<Feature>;   Optional: ResizeArray<Feature>
                 Alternative: ResizeArray<Feature>; Or: ResizeArray<Feature>         }
```

Ideally we would have wanted to generate this definition with the workbench as well, which would then eliminate this awkward conceptual mismatch between the grammar and the metamodel, but implementation-wise this would mean switching from F# to not just OCaml, but MetaOCaml [21], which will then allow to use staging to generate data structures instead of matching them with the specification. As it stands, we leave it for future work — or as an exercise for an eager reader.

The grammar of the Universal Variability Language in its final form looks as follows in Indentia:

```
Feature ::= [ goal => make ]
```

where Feature denotes the name of the type that we are parsing into, ⇒ represents sequential composition, [...] used for iteration. The `make` command, when witnessing a line, makes a new instance of the main type and adds it to the current context. For example, if the line says `Foo`, then it will be a new Feature with Name equal to "Foo". The `goal` command shifts the context deeper into the structure. For example, if the indentation increases after parsing `Foo`, the context moves inside it, and witnessing `mandatory` as a goal, readjusts the context to the Mandatory field inside Foo.

The transformation language Scripta is nothing as special as Indentia, it has statements of two kinds: `template` that defines a named template in terms of an interpolated string, and `each` ... ↦ ... `with` that applies a template to a type of data encountered anywhere within the model when transforming it to text. Generally speaking, interpolated strings by now are a fairly mainstream construct in software languages, even though they usually have more syntactically sound structure — e.g., `$"%s{tool};%s{modelName};%d{index};0;%s{phaseName};Time;%d{sw.ElapsedTicks*100L}"` in F# itself (this is how self-measurements are reported, according to the format requested in the case description [2]). Given the minimalistic nature of our implementation, we limited ourselves to words in upper case. Similarly, in model to text transformations other aspects like traversal strategies can play a major role, but we just assume recursive top down traversal with pre-ordered siblings. The script without the constraints part which is easy to guess, looks as follows:

```
template BEFORE ::= 'digraph FeatureModel {\nrankdir="TB"\nnewrank=true\nbgcolor="#1e1e1e"\
    nedge [color="white"]\nnode [style="filled" fontcolor="white" fontname="Arial Unicode MS,
    Arial" fillcolor="#ABACEA" shape="box"];\n'
template relation ::= '"SOURCE" -> "TARGET" [arrowhead="HEAD" arrowtail="TAIL" dir="both"]'
template boxed ::= 'NAME [fillcolor="#ABACEA" tooltip="Cardinality: None" shape="box"]'
each feature |-> boxed
each mandatory |-> relation with HEAD=dot TAIL=none
each optional |-> relation with HEAD=odot TAIL=none
each alternative |-> relation with HEAD=none TAIL=odot
each or |-> relation with HEAD=none TAIL=dot
template AFTER ::= '}'
```

As one can see, most of the space in this example is taken up by producing the right concrete syntax for DOT, and all the details about colours and arrow kinds were taken directly from the case description [2], essentially cloning them from the reference solution. This was done manually because ChatGPT would not be familiar with the Scripta syntax, which emerged while writing this example anyway.

5. Solution in Phases

The next subsections describe the solution in detail and may require some basic F# knowledge to fully appreciate them (F# skills can be easily substituted with OCaml skills and to some extent with Haskell skills). The subsections follow closely the same naming convention proposed by the case description [2]: the **Initialisation** phase is supposed to load the grammar/metamodel and the transformation specification; in the **Load** phase the solution loads a model; **Initial** phase performs the specified model transformation on the said model and produces an output; and the **Update** phase follows up by performing the same model transformation on other versions of the same model, if they are available, based either on new versions (as in our case) or on incremental deltas between the previous version with the incoming updates.

5.1. Initialisation

This phase is coded as one relatively simple function:

```
let Initialization() = (parseGrammar metaModelPath, parseTransformation transformationPath)
```

It is expressed in terms of two functions implementing the actual parsing. The first one is

```
let parseGrammar(filename: string) =
    let tokens = File.ReadAllText(filename).Split(' ')
    let transitions = Dictionary<string, string>()
    let firstStep,_,_ = parseStepChain transitions tokens 2
    { MainClass = tokens[0]; Start = firstStep.Value; Steps = transitions }
```

Let us take a moment to explain some optimisations. Conceptually everything is very easy, we take a string in the form of "left hand side ::= right hand side", take the left one since it is a string, and call another function to perform the actual heavy lifting in parsing the right hand side. Since we trivialise tokenisation, the right hand side is the list of tokens starting from the third one: skipping the left hand side which is one word, and "::=" which is just a terminal symbol. A fully implemented recogniser should check for this terminal correctness by string comparison, and a subsequent parser can just ignore it. Since we are cutting corners in this minimalistic implementation, we assume the input is correct, and simply ignore the second token.

The real question is: how to skip the first two tokens? The first version proposed by ChatGPT invoked `tokens[2..] |> Array.toList` which was eventually manually rewritten as marginally faster and definitely more FP-idiomatic `tokens |> Array.toList |> List.skip 2`. However, in an attempt to refactor the code to use faster arrays instead of more flexible lists, the corresponding helper function `parseStepChain` was rewritten as well to use string arrays and just pass current indices around, which is of course a major improvement since it eliminates both the type conversion and the head-tail deconstructions that used to happen at each step of the algorithm. Having this setup also removes the need for skipping anything in the list/array of tokens, since we can just ask the parsing function explicitly to start with the third position.

The impact of these optimisations on the actual performance of the tool are infinitesimal, since this is about parsing one line with seven tokens already separated by spaces. So, our explanation was not about huge gains in parser optimisations, but rather about the fact that in order to write nicer and faster code, one needs sometimes to be able to think across functions and juggle heterogeneous considerations about recursive algorithms and behaviour of data structure implementations. Since by definition vibe coding gives up on code comprehension by the human developer [11], and such optimisations are so spread out through code fragments and data definition, it can be claimed that at some very reachable level of complexity we will also have to give up on code comprehension by the LLM. Could it be that writing (or even generating) optimised code is the only thing that will be left over on one end of the spectrum once most development will move towards the vibe coding end?

```
let rec parseStepChain (dict: Dictionary<string, string>) (tokens: string[]) (startIdx: int) :
     string option * string option * int =
    let rec parseSequence idx (acc: string option) (past: string option) (prev: string option)
         : string option * string option * int =
        if idx >= tokens.Length then acc, past, idx
        else let token = tokens[idx]
             if token = "=>" then parseSequence (idx+1) acc past prev
             elif token = ")" || token = "]" then acc, past, idx+1
             elif token = "(" || token = "[" then
                 let first, last, remIdx = parseStepChain dict tokens (idx+1)
                 if prev.IsSome && first.IsSome then dict[prev.Value] <- first.Value
                 if token = "[" && last.IsSome && first.IsSome then dict[last.Value] <- first.
                     Value
                 parseSequence remIdx (if acc.IsNone then first else acc) (if last.IsSome then
                      last else past) None
             else
                  if prev.IsSome then dict[prev.Value] <- token
                  parseSequence (idx+1) (if acc.IsNone then Some token else acc) (Some token)
                      (Some token)
    parseSequence startIdx None None None
```

Model name	Versions	Length	Max Depth	Constraints	Max Complexity
automotive01	1	3314	25	2833	69
automotive02	4	15568..20670	21	666..1369	173..216
berkeleydb	1	107	15	20	258
busybox	37	441..633	3	463..681	136..268
financialservices	10	685..961	13	1001..1148	233..1321
linux	1	7486	15	3545	4080

Table 1
Some metrics computed on the models provided for solution evaluation: "Versions" is the number of versions that need to be processed in the Update phase (there is no Update phase if this number is 1); "Length" is the number of non-empty UVL lines in feature definitions; "Max Depth" is the maximal number of indentations used on one line, representing how deep in the tree that feature resides, and it is always an odd number by the nature of the UVL format; "Constraints" is the number of constraints in addition to the features; "Max Complexity" is the maximal number of characters used to define a constraint. For models with many versions we give the range if the metric returned varying values: they usually increase with each update, but not always, especially with the `financialservices` models which oscillate a lot along the update chain.

As we can see, it is the second function involved in the Initialisation phase that does the main heavy lifting. It is co-recursive with its helper function, and has three arguments that hold intermediate results, which is all quite classic and could have been written by a beginning first year functional programmer in any language. Parsing SCRIPTA is done line by line and also follows recognisable FP patterns like maps (not `map` but `iter` since we keep arrays as the base structure) and folds, as well as pattern matching and high level functions passed as arguments. Helper functions like `unquote` are also unremarkable even though the first version of it used to steal another 5 lines just for itself. The only detail truly worth mentioning is the replacement that takes place eagerly right here at the place where the template is called, and happens again at the last moment only for SOURCE and TARGET arguments. This is only one of the possible ways to implement interpolated strings, but at least in this case by far the most efficient one. The final implementation is a combined result of manual profiling, mental comprehension as well as discussions with ChatGPT.

```
let parseTransformation (filename: string):Transformation =
    let mutable templates = Dictionary<string,string>()
    let mutable iterators = Dictionary<string,string>()
    File.ReadAllLines(filename)
    |> Array.iter (fun line ->
            let parts = line.TrimEnd().Split(' ')
            match parts[0] with
            | "template" -> templates[parts[1]] <- unquote(line[14+parts[1].Length..])
            | "each" -> iterators[parts[1]] <- parts[5..] |> Array.fold (fun acc pair -> match
                 pair.Split("=") with | [|k; v|] -> acc.Replace(k, v) | _ -> acc) templates[
                parts[3]]
            | _ -> ())
    {Templates = templates; Iterators = iterators}
```

5.2. Load

The **Load** phase is supposed to load the model to be transformed. Depending on the interpretation, our solution could have been losing here in terms of performance, because it is definitely possible to generate very specific parser code with partly linear and partly even constant behaviour, and we are just running a simple interpretation of the grammar. However, the strict semantics enforced by the reference NMF solution, uses AnyText [22] to generate the parser before running it, and the time to generate it also counts against this phase. As a consequence, as we can see later in section 6, our solution fares quite well in terms of performance. In particular, the `linux` feature model seems to be challenging for NMF to handle while maintaining the speed. Looking at Table 1, we can hypothesise that the performance drop is due to the large number of constraints which are also the most complex ones. Our solution, after all, performs semi-parsing [23] and stores constraints as strings per line, and

the NMF solution does the full parsing at the Load phase and the pretty-printing at the Initial and Update phases.

```
let Load<'T> (grammar:MetaModel) (path:string) (make:string->'T) (goal:string->ResizeArray<'T>
    ->ResizeArray<'T>) : 'T =
    eprintfn $"Loading %s{path}"
    let mutable outOfFeatures : bool = false
    let contextStack = Stack<ResizeArray<'T>>()
    let mutable previous = 0
    let mutable step = grammar.Start
    let mutable context = ResizeArray<'T>(0) // fake start
    let mutable result = ResizeArray<'T>()

    for indent, content in ReadIndexedLines path do
        let delta = indent - previous
        previous <- indent
        if indent = 0 && result.Count <> 0 then outOfFeatures <- true
        elif outOfFeatures then addExtra(result[0],content)
        else
            if delta > 0 then
                if context.Count > 0 then contextStack.Push(context)
                if grammar.Steps.ContainsKey(step) then step <- grammar.Steps[step]
            elif delta < 0 then // && indent <> 0
                [1 .. (min -delta contextStack.Count)] |> List.iter (fun _ -> contextStack.Pop
                    () |> ignore)
                if delta % 2 <> 0 && grammar.Steps.ContainsKey(step) then step <- grammar.
                    Steps[step]
            if step = "goal" then context <- goal content context
            elif step = "make" then let feat = make(unquote(stripMeta(content)))
                                    context.Add(feat)
                                    if result.Count = 0 then result.Add(feat)
    result[0]
```

The code included above is pretty self-explanatory for those who have carefully read section 4. Not *all* corners have been cut here because we wanted to demonstrate at least with a couple of test cases that this newly emerged workbench is possible to use with different grammars. If this goal had disappeared, we could simplify away the function parameters used to create a new instance and to move the parsing context, as well as simplify the subtyping by dropping the type parameter from this function. In that scenario, the possible future work mentioned in section 4 of creating a staging MetaOCaml solution which generates the data structures as type definitions needed to hold the data coming out of the parser, would become almost unattainable altogether without a full rewriting of the entire implementation.

5.3. Initial

The **Initial** phase applies the already loaded transformation to the freshly loaded model, and outputs the result in the file where it belongs. We sacrifice two lines to create a helper function because it outweighs the pain of looking at the same condition four times within the same function.

```
let Initial (model:string) (feature:Feature) (script:Transformation) =
    let applyIfExists (script:Transformation) (name:string) (output:ResizeArray<string>) =
        if script.Templates.ContainsKey(name) then output.Add(script.Templates[name])
    let output = ResizeArray<string>()
    applyIfExists script "BEFORE" output
    if script.Iterators.ContainsKey("feature") then output.Add(script.Iterators["feature"].
        Replace("NAME", feature.Name))
    emitFeature script feature output
    if feature.Constraints.Count > 0 then
        applyIfExists script "BEFORE_CONSTRAINTS" output
        emitConstraint script feature output
        applyIfExists script "AFTER_CONSTRAINTS" output
    applyIfExists script "AFTER" output
    File.WriteAllLines(Path.Combine(modelDirectory, "results", $"{model}_{tool}.dot"), output)
```

It is assisted by this rather elegant function that adds all the necessary output lines for all four kinds of inner features any feature can have. This is a result of a manual rewrite, since ChatGPT proposed four bloated near-clones and failed to clean them up. It is often observed that LLMs are of significant help in optimising legacy systems and can have a boosting impact there [24] but this seems to be

less so for modern code written in relatively new languages. Perhaps what dictates applicability here, is a conceptual difference between restructuring low modernity [25] code which uses outdated or perhaps even deprecated code idioms, and performing performance-driven rewrites of already highly functioning and modern idiomatic code.

```
let rec emitFeature (script: Transformation) (feature: Feature) (lines: ResizeArray<string>) =
    let handleKind (kind:string, children:ResizeArray<Feature>) =
        if script.Iterators.ContainsKey(kind) then
            for child in children do
                lines.Add(script.Iterators[kind].Replace("SOURCE", feature.Name).Replace("
                    TARGET", child.Name))
                emitFeature script child lines
    ["mandatory", feature.Mandatory; "optional", feature.Optional; "alternative", feature.
        Alternative;  "or", feature.Or] |> List.iter handleKind
```

There are many ways to visualise constraints, but the case description asked for an HTML table, which is what is being generated here, according to the Scripta specification:

```
let emitConstraint (script: Transformation) (feature: Feature) (lines: ResizeArray<string>) =
    if script.Iterators.ContainsKey("constraint") then
        feature.Constraints |> Seq.iter (fun child -> lines.Add(script.Iterators["constraint"]
            .Replace("TEXT", makeSafeForSVG(child))))
```

It is unclear where does our solution lose execution time and why it is so much slower than the NMF solution. The code seems only to be doing lookups and string computations, so the reason is either one of those things, or something else, perhaps architectural or algorithmic, entirely. For the moment we call it a day, enjoy the implementation brevity and give up on performance.

5.4. Update

The **Update** phase implementation is profoundly uninteresting: after computing the name of the candidate file, it reloads and transforms it from scratch. There is no attempt to optimise it or to implement any kind of incrementality or diffing — so there is also no new knowledge about the possible feasibility to vibe code through it.

```
let Update (grammar:MetaModel) (script:Transformation) (name:string) (path:string) =
    let fix = path.Split("_01")
    if fix.Length = 2 then
        let rec processModels i =
            let nextModel = $"{fix[0]}_%02d{i}{fix[1]}"
            let nextName = $"{name}_%02d{i}"
            if File.Exists(nextModel) then
                measureTime "Update" i (fun() -> Initial nextName (Load grammar nextModel
                    makeFeature matchGoalFeature) script)
                processModels (i+1)
        processModels 2
```

The main reason for this defeatist approach is that it will entail two additional steps for which the architecture is not ready. First of all, the "git diff" format must be parsed — it is not a big deal, but will add another dozen of lines to the LOC count at least. Secondly and more importantly, the processing of "+"-lines should entail reconstructing the correct state of the parser (including the full stack of contexts) at the line where new content is added, and then proceeding with the parsing steps. To make matters worse, since we cannot trust "+"-lines to respect the indentation changes up to the point of only adding siblings to existing nodes, in some situations we would have to do a full reparse starting from the point of the end of the first positive diff. For "−"-lines it is even worse, since not only we need the state of the parser, but we need to find a way to undo parse actions, which in the architecture in its current form is not an option anyway.

To summarise, this solution would require significantly more lines to be written/generated, will impact memory footprint drastically due to the amount of data that needs to be stored, could only work if some smart corrections are brought in to adjust the architecture (i.e., caching the parser actions instead of executing them directly), and would only bring in reparsing benefits if there are only local changes happening reasonably late in the model. In other words, not worth it.

6. Solution Comparison

The code presented here, is far from perfect. It makes use of many assumptions that would be unacceptable in a serious software system which is meant to remain maintainable for decades to come. However, any randomly vibe-coded solution in whatever framework of whatever size, would also never satisfy this criterion.

The NMF solution provided for reference, is also not 100% polished and cuts at least some of the corners, like this function that matches on the type of the input, which should have been handled by built-in polymorphism of C# or at by a suitable design pattern:

```
private static void WriteConstraint(IConstraint constraint, TextWriter writer)
{
  switch (constraint)
  {
    case FeatureConstraint featureConstraint:
      WriteConstraint(featureConstraint, writer);
      break;
    case ImpliesConstraint impliesConstraint:
      WriteConstraint(impliesConstraint, writer);
      break;
    case OrConstraint orConstraint:
      WriteConstraint(orConstraint, writer);
      break;
    case AndConstraint andConstraint:
      WriteConstraint(andConstraint, writer);
      break;
    case EquivalenceConstraint equivalenceConstraint:
      WriteConstraint(equivalenceConstraint, writer);
      break;
    case NotConstraint notConstraint:
      WriteNotConstraint(notConstraint, writer);
      break;
    default:
      Console.Error.WriteLine($"Constraint type {constraint.GetType().Name} not supported.");
      break;
  }
}
```

When it comes to lines of code, the NMF [26] reference solution contains **58** LOC of AnyText [22] grammar, **48** LOC of NMeta [27] metamodel, **7878** LOC of C# generated from these two, and finally **508** LOC of handmade/vibed C#, which includes the model-to-text transformation since it is hardcoded.

Similarly, the EMF reference solution contains **49** LOC of UVL metamodel in EMF [28], **207** LOC of DOT metamodel in EMF, **23007** LOC of Java generated from these two, and finally **246** LOC of handmade/vibed Java, which is just the template to be extended.

In comparison to those, our solution contains **1** LOC of UVL grammar in Indentia, **13** LOC of UVL2DOT transformation in Scripta, and finally **150** LOC of handmade/vibed F#. There are 100-ish additional lines of F# with test cases that demonstrate that the parameters and languages are not just for the show, and it is possible to run the core of this solution with other grammars and other scripts.

Execution times are compared in Figure 1, memory consumption in Figure 2. The NMF solution is consistently outperforming the SLE solution on most phases, except for two very first runs of the Initialisation phase (the upstart of NMF seems to bring it up to 1.4 picoseconds while the upstart of SLE is barely 1 picosecond). The loading of the `linux` model seems to be a challenge for NMF as well, amounting to 1.3 picoseconds consistently across executions, while SLE is downwards of 0.1 picoseconds — as we discussed before, this is probably due to a large number of complex constraints used in that feature model. For memory usage, the SLE solution is consistently and uniformly consuming 2–4 times less memory that the NMF solution.

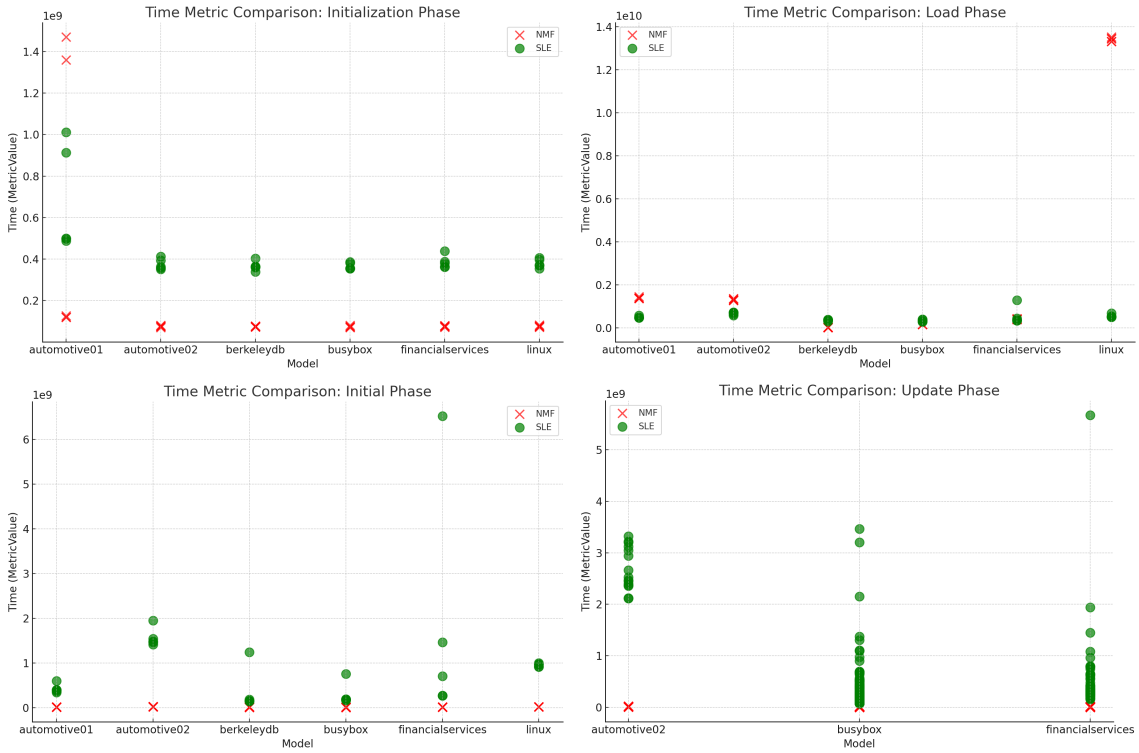

Figure 1: Time benchmarking of five runs of both fully working solutions across all available models: green circles for SLE (our solution), red crosses for NMF (reference solution). The `automotive01` models suffer from startup activities in the Initialization phase and take about 1400 ms per run the first two times for NMF and about 1000 ms per run for SLE; after that runtime of SLE halves but of NMF goes down tenfold — a decisive win for NMF. For the Load phase, both solutions are comparable except when it comes to the `linux` model on which the NNF solution surprisingly chokes. On the last two phases SLE loses, making NMF seem like an instantaneous action — being overall almost 200 times slower than NMF.

7. Conclusion

This work explored the design and implementation of a minimalistic, domain-specific transformation tool for translating UVL feature models into Graphviz DOT, created under the unique pressures and spirit of the TTC 2025 Live Contest. By combining lightweight language engineering with LLM-assisted vibe coding, we demonstrated that it is possible to achieve correct and competitive solutions outside traditional transformation frameworks — albeit with significant trade-offs in maintainability, extensibility, as well as in code comprehension. The key component to the success of this is a combination of human expertise and LLM boosts.

Our approach, leveraging the Indentia and Scripta mini-languages within a concise F# implementation, resulted in a tool that was notably compact and memory-efficient compared to established reference solutions, though it lagged behind in raw performance for some of the input models. The process also exposed some practical limits of current large language models in software language engineering tasks that demand custom parsing and non-trivial model transformations, and the paper includes a few reflection moments. In general, while AI suggestions provided useful scaffolding, human expertise and manual intervention were crucial to achieving a workable and idiomatic result.

Similarly to how *vibe coding* allows the practitioner to stay ignorant of the existence of the code, the process of *expert vibing* proposed in this paper, allows to have a firm unrelenting grasp on the conceptual architecture while staying if not oblivious then at least detached from the implementation details — at least until the optimisations need to happen there. This combines the power of letting the developer to stay in the flow [29] and generate executable code right where it is needed instead of leaving behind “TODO”s and other kinds of self-admitted technical debt.

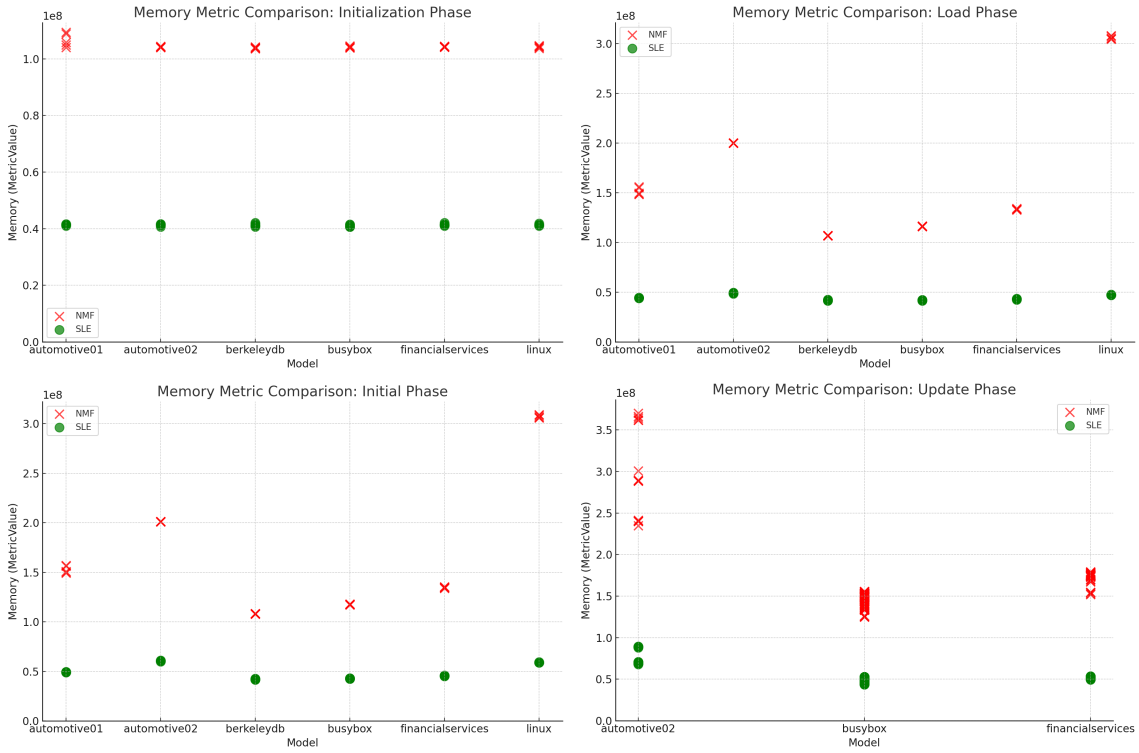

Figure 2: Memory consumption benchmarking of five runs of NMF (red crosses) and SLE (green circles). A decisive victory for SLE: at no single point of reference was this solution consuming more memory than the reference solution, usually being 2–3 times smaller. Footprints should be very comparable since both solutions rely on the same function of their shared platform: `Environment.WorkingSet`.

Ultimately, this experiment suggests that *expert vibing* — an iterative, intuition-guided workflow blending human insight and LLM assistance — can be a productive strategy for rapid prototyping in niche domains, but cannot (yet) substitute for principled engineering in projects that demand long-term maintainability or industrial strength. As the boundaries between generative intelligence and software development continue to blur, future work must investigate the interplay between language engineering, code generation, and expertise. In addition, there seems to be a deeper fundamental connection between indentation-based languages and a corresponding class of automata. This can be leveraged by compiling to maximally efficient parsing code instead of interpreting the grammar directly.

Quoting the inventor of the term *vibe coding*:

"Even vibe coding hasn't reached its final form yet. I'm still doing way too much." [11]

Declaration on Generative AI

Since the core premise of this work was to use GAI, the author used several models — mostly *GPT 4.1* with and without *Deep Research*, but also experimenting with *GPT 4o*, *GPT o1-mini*, *GPT o3* and *DeepSeek DeepThink*. Most code presented in this paper, was either partially generated by GAI tools, or written manually after witnessing GAI tools failing to generate it. After using these tools/services, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

During the writing phase of this work, the author used *GPT-4o* in order to generate Figure 1 and Figure 2 from the data he collected manually. After using the plot generator, the author reviewed the content as needed and takes full responsibility for the publication's content.

References

[1] G. Hinkel, S. Greiner, T. Le Calvar, A. Garcia-Dominguez, A. Boronat, F. Křikava, L. Rose, T. Horn, C. Krause, Transformation Tool Contest Live Contest, https://transformationtoolcontest.github.io, 2014–2025.

[2] G. Hinkel, S. Greiner, T. le Calvar, Universal Variability to Dot: The TTC 2025 Live Contest, https://github.com/TransformationToolContest/ttc2025-live/blob/main/docs/UVL_to_Dot_TTC_2025_Live_Contest.pdf, 2025.

[3] T. Horn, F. Křikava, L. Rose, Transformation Tool Contest Live Contest, https://transformationtoolcontest.github.io/2015/livecontest.html, 2015.

[4] A. Garcia-Dominguez, G. Hinkel, F. Křikava, Transformation Tool Contest: Solutions to the Social Media Live Case, https://transformationtoolcontest.github.io/2018/solutions_liveContest.html, 2018.

[5] D. Benavides, C. Sundermann, K. Feichtinger, J. A. Galindo, R. Rabiser, T. Thüm, UVL: Feature Modelling with the Universal Variability Language, Journal of Systems and Software 225 (2025) 112326. doi:10.1016/j.jss.2024.112326.

[6] J. Ellson, E. Gansner, Y. Hu, S. North, M. Jacobsson, M. Fernandez, M. Hansen, et al., Graphviz DOT Language, https://graphviz.org/doc/info/lang.html, 1991.

[7] G. Hedin, R. Hebig, V. Zaytsev (Eds.), Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering (SLE), Association for Computing Machinery, 2025. doi:10.1145/3732771.

[8] M. Fowler, Language Workbenches: The Killer-App for Domain Specific Languages?, MartinFowler.com, 2005. URL: https://martinfowler.com/articles/languageWorkbench.html.

[9] M. P. Ward, Language-Oriented Programming, Software: Concepts and Tools 15 (1994) 147–161. URL: http://www.gkc.org.uk/martin/papers/middle-out-t.pdf.

[10] K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools, and Applications, ACM Press/Addison-Wesley, 2000. URL: https://amzn.to/449LIEk.

[11] A. Karpathy, There's a new kind of coding I call vibe coding..., X, https://x.com/karpathy/status/1886192184808149383, 2025.

[12] M. E. Conway, How Do Committees Invent?, Datamation 14 (1968) 28–31. URL: https://www.melconway.com/Home/pdf/committees.pdf.

[13] P. McGuire, Pyparsing v3.2.3, https://pyparsing-docs.readthedocs.io/en/latest/index.html, 2024.

[14] E. Shinan, Lark 1.2.2, https://lark-parser.readthedocs.io, 2024.

[15] V. Zaytsev, Event-Based Parsing, in: T. Kamina, H. Masuhara (Eds.), Proceedings of the Sixth Workshop on Reactive and Event-based Languages and Systems (REBLS), 2019. doi:10.1145/3358503.3361275.

[16] V. Zaytsev, Parser Generation by Example for Legacy Pattern Languages, in: M. Flatt, S. Erdweg (Eds.), Proceedings of the 16th International Conference on Generative Programming: Concepts and Experience (GPCE), ACM, 2017, pp. 212–218. doi:10.1145/3136040.3136058.

[17] D. Leijen, P. Martini, A. Latter, et al., *Parsec*, Hackage, https://hackage.haskell.org/package/parsec; GitHub, https://github.com/haskell/parsec, 2006.

[18] F. Pottier, Y. Régis-Gianas, *Menhir*, https://gallium.inria.fr/~fpottier/menhir/; GitHub, https://github.com/LexiFi/menhir, 2005.

[19] C. Najmabadi, J. Parsons, H. Chang, T. Matoušek, S. Harwell, M. Vasani, J. Malinowski, et al., The .NET Compiler Platform ("Roslyn"), https://github.com/dotnet/roslyn, 2019.

[20] V. Zaytsev, A. H. Bagge, Parsing in a Broad Sense, in: J. Dingel, W. Schulte, I. Ramos, S. Abrahão, E. Insfran (Eds.), Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), volume 8767 of *LNCS*, Springer, 2014, pp. 50–67. doi:10.1007/978-3-319-11653-2_4.

[21] O. Kiselyov, MetaOCaml Theory and Implementation, 2023. arXiv:2309.08207.

[22] G. Hinkel, A. Hert, N. Hettler, K. Weinert, AnyText: Incremental, Left-Recursive Parsing and Pretty-Printing from a Single Grammar Definition with First-Class LSP Support, in: G. Hedin, R. Hebig, V. Zaytsev (Eds.), Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering, SLE, Association for Computing Machinery, 2025, p. 98–111. doi:10.1145/3732771.3742716.

[23] V. Zaytsev, Formal Foundations for Semi-parsing, in: S. Demeyer, D. Binkley, F. Ricca (Eds.), Proceedings of the Software Evolution Week (IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering), Early Research Achievements Track (CSMR-WCRE 2014 ERA), IEEE, 2014, pp. 313–317. doi:10.1109/CSMR-WCRE.2014.6747184.

[24] V. Singh, C. Korlu, W. K. G. Assunção, Experiences on Using Large Language Models to Re-Engineer a Legacy System at Volvo Group, in: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2025, pp. 102–112. doi:10.1109/SANER64311.2025.00018.

[25] C. Admiraal, W. van den Brink, M. Gerhold, V. Zaytsev, C. Zubcu, Deriving Modernity Signatures of Codebases with Static Analysis, Special Issue in the Journal of Systems and Software: Open Science in Software Engineering Research (JSS) 211 (2024). doi:10.1016/j.jss.2024.111973.

[26] G. Hinkel, NMF: A Multi-platform Modeling Framework, in: A. Rensink, J. Sánchez Cuadrado (Eds.), Theory and Practice of Model Transformation, Springer International Publishing, 2018, pp. 184–194. doi:10.1007/978-3-319-93317-7_10.

[27] G. Hinkel, et al., NMeta, https://nmfcode.github.io/models/NMeta.html, 2024.

[28] E. Merks, D. Huebner, et al., Eclipse Modeling Framework, https://eclipse.dev/emf/, 2006.

[29] S. Janssens, V. Zaytsev, Go with the Flow: Software Engineers and Distractions, in: T. Kühn, V. Sousa, S. Abrahão, T. C. Lethbridge, E. Renaux, B. Selić (Eds.), MoDELS'22 Companion Proceedings: Sixth International Workshop on Human Factors in Modeling / Modeling of Human Factors (HuFaMo), 2022, pp. 934–938. doi:10.1145/3550356.3559101.