Can AI Make the Rules?
A Pipeline for Generating Graph Transformation Systems

Reiko Heckel Muhammad Saad

School of Computing and Mathematical Sciences
University of Leicester

MODELSWARD 2026

Can AI Make the Rules?
A Pipeline for Generating Graph Transformation Systems

Reiko Heckel Muhammad Saad

School of Computing and Mathematical Sciences
University of Leicester

MODELSWARD 2026

Spoiler: LLMs help write rules; executable models make them reviewable, deterministic, auditable

Outline

Regulated (= rule-based) domains

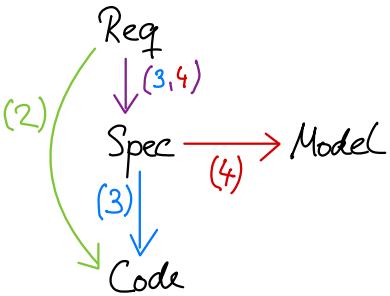
In domains like tax, law, compliance, insurance:

- **Correctness** is essential and needs to be verifiable
- **Accountability** requires auditable artefacts, not just plausible outputs.
- **Reproducibility** matters: same inputs should yield same results.
- **Evolvability** matters: rule changes must be localised and traceable.

Problem: LLMs are good at NLP, getting better at informal reasoning, but unreliable logic engines.

Four roles of LLMs in software

1. **(1)** Direct problem solving (LLM call, agents)
   Opaque, non-deterministic, hard to debug.
2. **(2)** LLM-generated code ("vibe coding")
   Deterministic after freezing code,
   but regeneration unstable; no traceability
3. **(3)** Specification-driven generation (SDD)
   More structure, but semantic gap
   risks requirements "drift"; little traceability
4. **(4)** Executable model generation (e.g., GTS)
   LLM assists formalisation; execution
   delegated to a fixed semantics engine.



Executable models provide the *semantic anchor*.

Outline

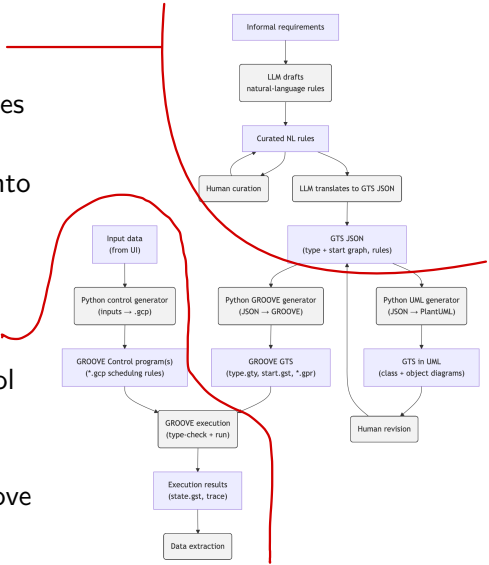
Pipeline overview: design decisions

LLMs at design time

- drafting NL rules
- translating curated rules into Intermediate representation (JSON)

LLM-free runtime

- generate control program from input
- execute in Groove
- extract output



From the JSON IR

- **PlantUML** diagrams: for human review
- **GTS** artefacts: for machine execution
- control program: deterministic execution

Human effort where it is effective:

- Curation of NL rules
- Review of generated UML diagrams
- Revision of IR

Typed graph transformation systems (GTS)

GTS as visual model:

- Type graph: node/edge types; attributes and their domains
- Start graph: initial state template, conforms to type graph
- Rules: visual, local, declarative, explicit state changes
  - create / preserve / delete nodes and edges,
  - update attribute values via expressions,
  - application conditions (guards, constraints)
  - parameters and their bindings

Why graphs?
- object data model
- knowledge graphs
- diagrams

And semantic anchor:

- Formal, executable semantics of rule application
- Auditable, every state change attributable to a named application
- Deterministic if needed, via explicit control program, or as nondeterministic safety envelop

Outline

Tax case study: from informal requirement to curated NL rule

Informal requirement:

> *Self-employment profit is revenue minus allowable expenses; the model must represent each income source explicitly and compute the net amount used by subsequent totals and allowances.*

Drafted + curated NL rule:

> *IF a person has income from self-employment THEN create an income source linked to that person, setting net_income = revenue - expenses.*

This fixes vocabulary and parameter discipline before formalisation.

IR (JSON): rule structure

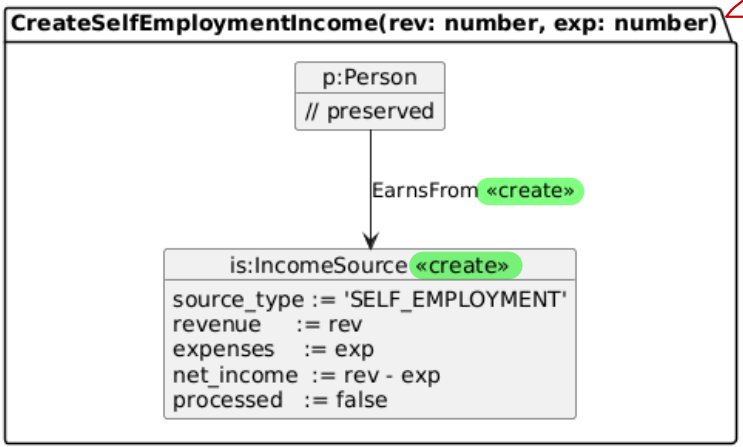
Explicit definition of create/preserve/update effects, parameters, conditions

Excerpt in structured text

- `name`: `CreateSelfEmploymentIncome`
- `parameters`: `rev:number`, `exp:number`
- `nodes`:
  - preserve `p:Person`
  - create `is:IncomeSource` with updates: `revenue:=rev`, `expenses:=exp`, `net_income:=rev-exp`
- `edges`: create `EarnsFrom(p,is)`
- `applicationConditions`: guard / NACs if needed

IR is checkable (JSON schema, type consistency), versionable, and generator-friendly.

PlantUML visualisation

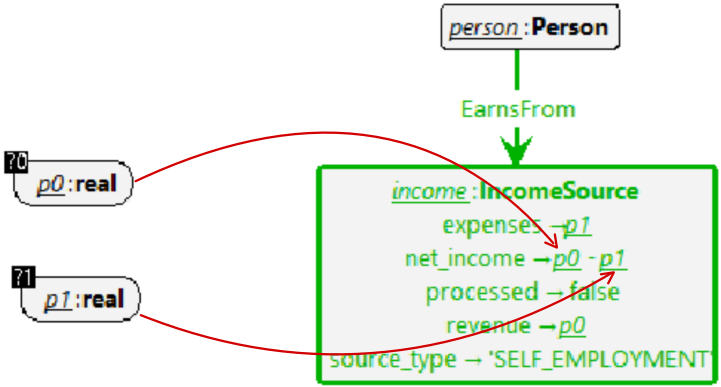


Rule names with parameters

integrated rule graph

Generated by Python script, which was generated by a coding agent

Visual, executable GROOVE rule



Generated by Python from JSON IR, using the GROOVE's GXL format.

Direct generation by LLM of rules in GXL is hard, hence the use of JSON IR.

GROOVE control program

Instantiate inputs, then try calculation rules in statutory order.

Control program (sketch)

```
<
  CreateSelfEmploymentIncome(50000.00, 0.00);
  CreateDividendIncome(0.00);
  CreateEmploymentIncome(0.00);
  CreateSavingsInterestIncome(0.00);
  inputParams(0.00, 0.00, false, false);
>
...
try { rule10_(); } else { }
...
```

Generated by Python from a template and inputs.

Deterministic execution + auditability: trace of named rule applications with actual parameters.

Outline

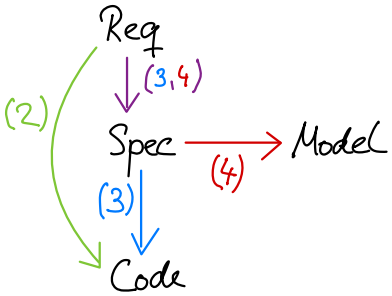
1. LLMs in regulated domains
2. Pipeline for rule generation and validation
3. Example walk-through: a tax calculator
4. Experimental comparison
5. Takeaways and next steps

Experimental setup

Domain: fragment of UK personal income tax computation.

Approaches compared:

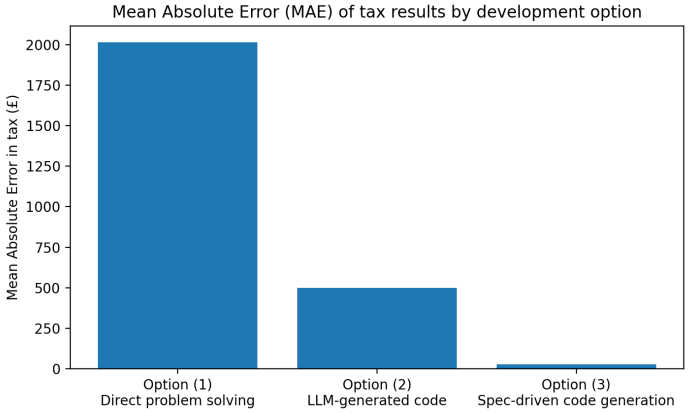
- Option (1): runtime LLM calls
- Option (2): LLM-generated Python ("vibe coding")
- Option (3): spec-driven code generation
- Option (4): executable GTS via pipeline (reference)



Evaluation dimensions:

- correctness vs reference (MAE over scenarios),
- transparency / traceability,
- determinism and reproducibility.

Results: Mean Absolute Error (MAE) across options



MAE of Options (1)–(3) as deviation from GROOVE model (4) across diverse scenarios

- income levels near allowance thresholds,
- transitions between tax bands, combinations of income sources, deductions, etc.

Correctness, transparency, reproducibiility

Option (1) worst MAE: implicit, unstable internal semantics

- errors impossible to systematically debug

Option (2) improved MAE: deterministic after freezing, but opaque

- systematic errors: misinterpretations applied consistently,
- review requires low-level code inspection; rules are entangled.

Option (3) lowest MAE: spec extension / drift

- extra assumptions incorporated into spec,
- higher MAE not due to failure, but because (3) is "more correct" than reference model (4)

Option (4): discrepancies *tangible at rule level*

- formally defined rule set: explicit, versioned, reviewable,
- deterministic execution under a fixed semantics with execution traces

Outline

1. LLMs in regulated domains
2. Pipeline for rule generation and validation
3. Example walk-through: a tax calculator
4. Experimental comparison
5. Takeaways and next steps

Where LLMs help, and where they don't

Use LLMs for accelerating the design pipeline to executable models

- drafting requirements and candidate rules,
- translating curated rules into a structured IR,
- generating guardrails and rule-based agents

Do not use LLMs for critical runtime or hard-to-review code

- executing rules implicitly when accountability is required,
- being the sole arbiter of rule ordering and edge-case behaviour,
- producing opaque code, post-hoc narratives

AI can help make the rules. But we need executable, human-readable models to run them.

Toolchain pattern

Pattern

NL Requirements → Curated rules → JSON IR → Executable model + traces

IR as DSL that:

- makes LLM output *structurally constrained* and checkable,
- enables multiple consistent views (UML, GROOVE),
- supports regression testing and diff-friendly evolution at the rule level.

JSON IR for GTS joint effort with Gabi Taentzer + colleagues in Marburg.

Future work: applications and pipeline

Beyond the tax case study:

- apply to other regulated domains (law, insurance, compliance).

Reduce manual effort in formalisation:

- tighter tool integration across the pipeline,
- incorporate feedback from testing/analysis to guide rule generation,
- improve human-in-the-loop workflows for curation and revision.

Goal: AI modelling agents for GTS.

Future work: community

Tool support / engineering directions:

- Stabilise and standardise JSON IR for rule-based executable models.
- Generator libraries:
  - IR $\rightarrow$ diagram views (for review),
  - IR $\rightarrow$ executable artefacts (multiple backends),
- Validation and QA:
  - schema/type checking of IR,
  - analysis tools,
  - test suites and data sets.

Established MDD technology now much easier to build using coding agents!

What about agents?

Use AI agents (based on runtime LLMs) for:

- generating solutions that can be formally verified
  - planning based on domain knowledge: LLM
  - checking legality of steps and sequences: safety envelop, can be AI generated
- natural language tasks
  - NL UI
  - NL documents $\leftrightarrow$ structured data (IR, DSL)
- build in traceability by logging and documentation

And LLMs are not the only AI in town ...