

Rules as Code for Adaptable, Accountable and Reusable Software

L. Thomas van Binsbergen – University of Amsterdam

Rules as Code (RaC) is the practice of encoding legal rules, policies, and norms ('rules') as machine-readable and sometimes machine-executable specifications ('code'). The approach is used by governmental organisations to scrutinize rules and to communicate to citizens about their rights. In this article, I argue that in addition to transparency and scrutiny, RaC can form the basis for automated governmental services implemented in *adaptable* and *accountable* software. Moreover, owing to the *reuse* of specifications and services, the approach has the potential to scale up.

The thesis is that rule specifications can play a central role in the engineering of compliant software, both as part of the design, running and auditing of the software. With a sufficient separation of concerns, adaptability follows from the ability to extend and modify rules without redesigning the systems in which the rules integrate. Accountability follows from explicating and recording the rules and facts used in compliance decisions made by the system. Reuse can be achieved by identifying the legal and computational concepts that can intermediate between specification and implementation.

This article is a follow-up to my presentation at the 2026 edition of the [Rules of Code Europe conference](#). The slides of contributions mentioned in this article can be found below [0].

Rules as Code in Context

The Rules as Code approach stems from the observation that laws and norms can be re-structured to a machine-readable format. This formalisation process requires making certain implicit information explicit. Together, lawyers provide context-sensitive interpretations of the rules and software engineers explicate the relations between elements of the rules. As a consequence, ambiguities, incompleteness, conflicts, and open-textured terms surface naturally in the formalisation phase, rather than resulting in disputes later. Moreover, the resulting scrutiny can prompt feedback to lawmakers, potentially resulting in rules that are easier to operationalise.

Crucially, rules defined as code are not merely technical. They form a shared artefact for co-development by domain experts. This is typically achieved by preserving the natural language of the original normative sources in combination with the notational clarity required for software. Legal experts scrutinise the faithfulness and justifiability of the encoding; software experts assess logical correctness and establish the integration within software services. Together, the experts develop test scenarios to validate both the legal and computational outcome of applying the rules.

The approach is nicely exemplified by the RuleSpeak (RegelSpraak) language [1,2], developed by the Dutch Tax Administration to formalise Dutch tax law. A controlled subset of Dutch is used to encode computations, logical expressions and knowledge derivation rules. The rules are encoded by choosing from a set of specific sentence structures that each establish a formal, unambiguous connection between the subjects and objects mentioned in the rules. The ALEF engine supports error-checking RuleSpeak rules and processing real and hypothetical cases.

The Rules as Code Europe conference showed that the approach is acquiring momentum across Europe with a wide array of languages, tools, and services having been showcased. At

present, *the technical approaches are mostly isolated*, demonstrating a tight coupling between the service, the language and the reasoning capabilities used to deliver the service. However, previous research I conducted suggests that a decoupled approach is possible.

A Vision towards Reusable Reasoning Capabilities

My research investigates the following simple hypothesis:

There exists an intermediate language that can encode arbitrary normative sources as executable rules for delivering a multitude of compliance-related services.

The language is ‘intermediate’ in the sense that it is neither catered to a specific use case or sub-domain nor to a specific service. Technically, a domain-specific RaC-editor can translate its rule-representation into a specification of the intermediate language, target one or more of the reasoning engines supported by the intermediate language, and integrate the result into a domain-specific service.

To support multiple ‘front-end’ languages, the intermediate language needs to include general normative concepts such as prohibition, obligation, and power. Other requirements can be derived from the wealth of normative languages that currently exist. To support multiple ‘back-ends’ and services, the language must support multiple computational semantics, each defining precisely certain analyses performed by reasoning engines.

The hypothesis has been explored through two related domain-specific languages (DSLs) developed at the University of Amsterdam: FLINT [3] and eFLINT [4].

- **FLINT** is a front-end language aimed primarily at legal and policy experts. It foregrounds natural language text and explicit links to normative sources. The goal is to let domain experts structure and annotate rules while retaining the original legal wording, facilitating traceability and legal validation. In combination with version management, FLINT specifications can be reused across applications.
- **eFLINT** is the machine-executable back-end language. It gives a computational meaning to the concepts used in FLINT, enabling various forms of automated reasoning. For example, eFLINT can be used to analyse scenarios, to verify business processes and to search for scenarios satisfying certain (un)desirable properties [5]. The latter can be used for abduction-style reasoning as is also supported by Blawx [6].

This multiplicity of reasoning modes is crucial: it admits automation throughout the software development lifecycle, supporting compliance-by-design, runtime compliance checking, and auditing. Interestingly, all modes share a relatively simple interface consisting of a set of rules, a partial or complete scenario, and a query. This opens the way to a general approach to developing adaptable and accountable services on top of a set of reusable reasoning capabilities.

Demonstrating Accountability and Adaptability in DMI

Together with partners in the Dutch Metropolitan Innovations (DMI) ecosystem [7], we have been able to demonstrate reuse, adaptability and accountability in the development of the Products and Services eXchange (PDX) of DMI. We applied the Rules as Code approach to formalise parts of the Ecosystem Rules (‘Het Afsprakenstelsel’), including relevant GDPR (AVG) provisions concerning the lawfulness of processing personal data. The resulting rules, expressed in eFLINT, are integrated into the Governance component of the PDX. More

information about the PDX can be found here [8]. The formalisation of the Ecosystem Rules can be found here [9].

The Governance component supports a negotiation between an organisation making data available and an organisation consuming data. Together they must reach an agreement about for which purposes, and under which conditions, a dataset may be used. The negotiation is driven by the following reasoning process:

- Based on the Ecosystem Rules, questions are asked to establish facts and to reason about the compliance of the agreement currently being formed. For example, the data provider is asked if the offered data contains personal data. If so, the data provider is asked for the legal basis needed to be established lawfulness according to the GDPR.
- The data provider can set additional usage conditions on the use of its data, which are included as additional rules, potentially prompting further questions to the data provider or data consumer. For example, a condition stating that the data consumer must be a knowledge institute.
- Questions are asked until sufficient information is provided to draw a conclusion.

This process requires two reasoning capabilities: (a) establishing whether more information is required and (b) determining whether the current scenario results in a violation. An agreement can only be formed if the scenario is not violating. A positive scenario assessment can only follow if all required information is provided.

Accountability of the system is achieved by recording the rules and the facts required to determine whether an agreement can lawfully be formed. This makes it possible to reconstruct how it was concluded that an agreement could be formed. The agreement itself includes references to the rules encoding the usage conditions.

Adaptability of the system is achieved because the rules are maintained as a separate, modular rule-specification. The rules can be extended or modified without requiring a drastic redesign of the PDX architecture or protocols. New regulatory requirements, ecosystem rules, or sector-specific constraints can be added by augmenting the rule base, benefiting from the inherent extensibility and modularity of eFLINT specifications.

Integration patterns

The governance component of the PDX is an example of integrating rules as code within a so-called expert system. Other approaches to integrate a rule-specification exist. For example, rules can be used as policies within a policy decision point (see [10]) for making access control decisions. Formal verification can be used to verify the compliance of a business process model, which itself can be translated to executable code using model-driven engineering [11]. Techniques from runtime verification can be used to monitor compliance as a system is running and process mining admits checking compliance based on event logs [12]. That we have been able to demonstrate several of these integrations with eFLINT provides evidence in support of our hypothesis.

Outlook

The work presented at Rules as Code Europe 2026 sits at the intersection of legal theory, computer science, and software engineering. By treating rules as executable code with carefully designed semantics, we can move beyond using RaC for interpretative transparency. In

addition, we can develop infrastructures like DMI-PDX that demonstrate reuse, adaptability and accountability.

The long-term research objective is to generalise from our lessons learnt with FLINT and eFLINT and develop the hypothesised intermediate language. The experiments with FLINT, eFLINT, and the operationalisation efforts within DMI suggest that this goal is within reach.

References

- [0] Rules as Executable Code for Adaptable and Accountable Software. L. Thomas van Binsbergen. 2026. [\[Link\]](#)
- [1] RegelSpraak: a CNL for Executable Tax Rules Specification. Mischa Corsius, Stijn Hoppenbrouwers, Mariette Lokin, Elian Baars, Gertrude Sangers-Van Cappellen, Ilona Wilmont. 2021. [\[Link\]](#)
- [2] Specificaties RegelSpraak. Belastingdienst. 2025. [\[Link\]](#)
- [3] The FLINT Ontology: An Actor-Based Model of Legal Relations. Jeroen Breteler, Thom van Gessel, Giulia Biagioni, Robert van Doesburg. 2023. <https://doi.org/10.3233/SSW230016>
- [4] eFLINT: A Domain-Specific Language for Executable Norm Specifications. L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. 2020. [\[Link\]](#)
<https://doi.org/10.1145/3425898.3426958>
- [5] A Stable Model Semantics for eFLINT Norm Specifications and Model Checking Scenarios. Christopher Esterhuyse, Tim Müller, L. Thomas van Binsbergen. 2025. [\[Link\]](#)
<https://doi.org/10.1145/3742876.3742882>
- [6] Blawx | User-friendly Rules as Code online. Jason Morris. 2022-2026. [\[Link\]](#)
- [7] DMI Ecosystem. [\[Link\]](#)
- [8] The Products and Data eXchange (PDX) subsystem of the DMI Ecosystem. 2025-2026. [\[Link\]](#)
- [9] Formalisation of the governance rules ('Afsprakenstelsel') of the DMI Ecosystem. Tim Müller. 2025. [\[Link\]](#)
- [10] Dynamic generation of access control policies from social policies. L. Thomas van Binsbergen, Milen G. Kebede, Joshua Baugh, Tom van Engers, Dannis G. van Vuurden. 2022.
<https://doi.org/10.1016/j.procs.2021.12.221>
- [11] Model-Driven Engineering. Douglas C. Schmidt. 2006. <https://doi.org/10.1109/MC.2006.58>
- [12] Conformance Checking. Wil M. P. van der Aalst. 2011. https://doi.org/10.1007/978-3-642-19345-3_7