A Stable Model Semantics for eFLINT Norm Specifications and Model Checking Scenarios

Christopher A. Esterhuyse
c.a.esterhuyse@uva.nl
University of Amsterdam
Amsterdam, Netherlands

Tim Müller
t.muller@uva.nl
University of Amsterdam
Amsterdam, Netherlands

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
University of Amsterdam
Amsterdam, Netherlands

Abstract

Since its introduction at GPCE2020, the eFLINT norm specification language has been used in academic and industrial applications to specify and automate compliance for various norms, such as privacy regulations and data processing agreements. The eFLINT interpreter has been used to automate the analysis of real-time or historical cases by computing logical consequences and reporting normative violations.

To support future language and tooling developments, we contribute a formal definition of the language as a translation to first-order logic programming with stable model semantics. The described semantics aligns with the previous semi-formal descriptions of the language, but resolves issues relating to logical inference with negative antecedent and aggregation operators. Specifically, we formalise the connection between eFLINT's derivation rules and Horn clauses under the stable model semantics. Secondly, by repurposing the Clingo answer-set solver as a highly-optimised eFLINT interpreter, we extend the toolset for eFLINT with model-checking abstract properties in addition to case analysis.

We evaluate the new semantics and interpreter via an empirical comparison of the existing implementation to our prototype implementation. We observe that the expected subset of our tests have the equivalent behaviours.

CCS Concepts: • **Applied computing** → *Law*; *Enterprise modeling*; • **Software and its engineering** → **Domain specific languages**; *Semantics*; • **Theory of computation** → *Constraint and logic programming*; *Automated reasoning*.

Keywords: norm, specification languages, dynamic semantics, logic programming, model checking, answer set solving

ACM Reference Format:
Christopher A. Esterhuyse, Tim Müller, and L. Thomas van Binsbergen. 2025. A Stable Model Semantics for eFLINT Norm Specifications and Model Checking Scenarios. In *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '25), July 3–4, 2025, Bergen, Norway.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3742876.3742882

GPCE '25, Bergen, Norway

ACM ISBN 979-8-4007-1995-0/25/07
https://doi.org/10.1145/3742876.3742882

1 Introduction

The eFLINT language, first introduced at GPCE2020, is a domain-specific specification language designed to enable (static and dynamic) reasoning with interpretations of normative documents such as laws, regulations, and contracts [34]. eFLINT has been used in research of normative multi-agent systems [24, 37], distributed data processing systems [9, 16, 32, 33], and to develop partially automated data governance solutions [2, 35]. In these cases, the language has been used to specify, for example, data processing and consortium agreements, data access or usage conditions, and parts of the European *General Data Protection Regulation* (GDPR) [10].

From the start, eFLINT was designed to bridge the gap between normative concepts typically expressed by legal experts, and event-driven software systems requiring legal regulation. Its high level of abstraction lets eFLINT serve as a *lingua franca* for norms from various domains. For example, once encoded in eFLINT, smart contracts, EU data protection regulations, and organisational policies have meaningful compositions [33]. The *eFLINT interpreter*, available online[1], checks compliance of a given eFLINT *scenario* against a given eFLINT specification. Scenarios instantiate specifications by laying out the events that happen(ed). Precisely, the tool computes logical consequences and enumerates the normative *violations*: unmet obligations and prohibited actions.

Section 2 explains the eFLINT language in more detail. Until then, we exemplify the typical usage of the language; the eFLINT interpreter monitors the compliance of a software system by checking system traces (modelled as scenarios) for compliance against regulatory norms (modelled as a specification). Example 1 introduces our running example of such a usage; in this case, the specification captures the norm that data users must inform data controllers of (the purpose of their) data accesses within a given time frame.

Example 1 (Running Example eFLINT Usage)**.** *A system is regulated by the following specification, which expresses: accessors of datasets must notify the dataset's controllers (or the administrator by default) within ten time steps of the access.*

[1]The Haskell implementation sees active development. But we focus on the version authored on the 13th of February 2025: https://gitlab.com/eflint/haskell-implementation/-/blob/cfe73e9a0ed594e384027ea29529da62dab1fe0f

```
// These lines define three basic data types
Fact instant Identified by Int
Fact user    Identified by String
  // 'controller' is defined as a 'user' alias soon
  // For each controls that holds, its controller holds.
  Derived from (Foreach controls: controls.controller)
Fact dataset Identified by String
  Derived from (Foreach controls: controls.dataset)

// Introduces aliases for users and instants
Placeholder controller For user
Placeholder deadline   For instant

// (Mutable) relations over users, instants, and datasets
Fact elapsed Identified by instant
Fact controls Identified by controller * dataset
  // For each dataset that holds, the admin controls it
  // if there exists no non-admin user that controls it.
  Derived from (Foreach dataset:
    controls(user("Admin"),dataset)
    Where Not(Exists user:
      user != user("Admin") && controls(user,dataset)))

Act access Actor user Related to dataset, instant
  Derived from (Foreach user, controls, instant:
    access(user, controls.dataset, instant))
  // When triggered, the access has these effects
  Creates (Foreach controls:
    must_notify(user, controls.controller,
      access(user,dataset,instant), instant + 10)
    Where controls.dataset == dataset)

// Facts, but their elements are conditionally violated
Duty must_notify Holder user Claimant controller
  Related to access, deadline
  // Predicates when these duties are violated
  Violated when Holds(elapsed(deadline))

Act notify Actor user Recipient controller
  Related to must_notify Terminates must_notify
  Derived from (Foreach must_notify: notify(
    must_notify.user,must_notify.controller,must_notify))
```

The following eFLINT scenario models a hypothetical system configuration just after user Bob accesses the X-Ray dataset.

```
+dataset("X-Rays").
+controls(user("Amy"),dataset("X-Rays")).
+instant(9).
+user("Bob").
access(user("Bob"), dataset("X-Rays"), 9).
```

Given the above input, the eFLINT interpreter emits the following output, showing the consequences of each scenario step in turn: derived created (derived) instances, and that Bob's access action is ultimately permitted (enabled).

```
+dataset("X-Rays").
| +user("Admin")
| +controls(user("Admin"),dataset("X-Rays"))
+controls(user("Amy"),dataset("X-Rays")).
| user("Amy")
+user("Bob").
+instant(9).
| +access(user("Admin"),dataset("X-Rays"),instant(9))
| +access(user("Amy"  ),dataset("X-Rays"),instant(9))
| +access(user("Bob"  ),dataset("X-Rays"),instant(9))
access(user("Bob"),dataset("X-Rays"),instant(9)) ENABLED
| +notify(user("Bob"),user("Admin"), ... ,instant(19)))
| +must_notify(user("Bob"),user("Admin"), ... )
| +notify(user("Bob"),user("Amy"), ... ,instant(19)))
| +must_notify(user("Bob"),user("Amy"), ... ,instant(19))
```

The eFLINT language is motivated, described, and demonstrated in research articles [33, 34]. The syntax and fundamentals of the semantics are defined using conventional mathematical notation and precise natural language. For example, [34] makes clear that each specification denotes an action-labelled transition system, and that each scenario denotes a sequence of actions that traces a path through the transition system. Figure 1 visualises a scenario.

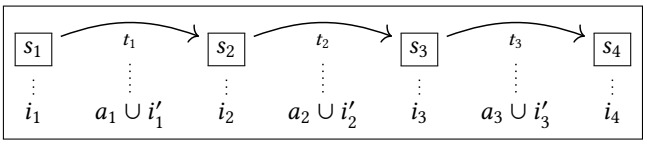


Figure 1. A trace through states s_{1-4} over transitions t_{1-3}, induced by a scenario triggering actions $[a_1, a_2, a_3]$ in sequence. Each state and transition models the (current) normative relationships as (elements of) some structured data base i.

However, at time of writing, some language features are not rigorously defined. For example, the eFLINT description in [34] suffices for users to agree that the syntactic term `Derived from (Foreach dataset: ...)` in Example 1 denotes an eFLINT *derivation clause* which gives the administrator control of datasets which have no other controllers. But some details remain unclear. Does the administrator retain control over the X-Rays once Amy has successfully claimed control? In the original interpreter, querying `?controls("Admin","X-Rays")` after Example 1 yields `query successful`; the administrator appears to retain control. But this behaviour conflicts with the plausible interpretation of what is documented in [34]: derivation clauses should apply (only) when their conditions are satisfied. The identified issue is that negation (`Not`), universal quantification (`Forall`), and aggregation (*e.g.*, `Max`) exhibit unspecified (thus unexpected) behaviour. For example, the outcome of the query is flipped if `user` is named `agent` instead.

In this article, we (re)define the eFLINT semantics. Our goal is to formalise eFLINT's existing semantic concepts in detail, such that eFLINT scenarios and specifications have unique and unambiguous meanings. We select an intuitive and mature foundation: first-order logic programming under the stable model semantics [13]. This is also pragmatic choice, as it lets us leverage existing logic programming tools. Precisely, we define the semantics of eFLINT with a translation to the *Clingo* language, *i.e.*, the input language of the Clingo answer-set solver tool. Thus, eFLINT inherits the clarity of the Clingo semantics, and we repurpose the Clingo solver as our new eFLINT interpreter.

We recreate the existing eFLINT descriptions and interpreter as much as possible. Specifically, we realise the description in [34] by aligning key eFLINT constructs with the inference rules (Horn clauses) of logic programming. Thus, users are afforded powerful, (de)compositional reasoning

about scenarios and systems via specifications. As part of this (re)design, we make two changes relative to the existing eFLINT interpreter. Firstly, the ambiguity of the original description regarding the semantics of derivation clauses with negation, iteration and aggregation is resolved. We do so by connecting these constructions to the literature on (the challenges of) reasoning with negated conditions in logic programming. In the running example, the result is that control over a dataset is exclusive to controllers or to the administrator (if there are no controllers). Secondly, we leverage our translation to Clingo to support a new, more general use case for eFLINT: via Clingo, our tool *searches* for scenarios with user-defined properties. For example, 'find all counterexamples to the proposition: scenarios taking only permitted actions violate no duties'. This use case has many practical applications, *e.g.*, in model-checking and recommendation, and is already popular with languages related to eFLINT, such as Symboleo [25] and FIEVEL [36].

We formulate our translation from eFLINT to Clingo as the multi-stage translation pipeline shown in Figure 2. This helps to structure our presentation. But moreover, it introduces novel intermediate representations which are useful in their own right. For example, *Core eFLINT* is entirely embedded in Clingo, but it captures much of th eFLINT semantics, and unifies eFLINT specifications with scenarios.

Concretely, this article makes the following contributions:

- A translational semantics for eFLINT specifications and scenarios with first-order logic programming and stable model semantics as the semantic domain.
- An implementation of our semantics that leverages the Clingo solver for reasoning.
- A Clingo-embedded, domain-specific, core language for reasoning with dynamic facts, acts and violations.
- An evaluation of contributions by confirming the expected input/output (in)equalities of the interpreters.

Our artefact [19] at https://doi.org/10.5281/zenodo.15188959 includes our implementation and experimental data.

This article proceeds as follows. Sections 2 and 3 give accounts for eFLINT and Clingo, the source and target languages of our translation, and their respective background literature. Sections 4 to 7 build up the translation stages 'bottom-up', *i.e.*, we build layers of abstraction atop Clingo. The stage in Section 7 handles the two forms of reasoning: scenario-compliance and scenario-search. Section 8 evaluates our contributions, and Section 9 identifies related and future work, before Section 10 concludes.

$$\textit{spec.} \times \textit{scenario} \Rightarrow \textit{eFLINT core} \rightarrow \textit{state trace} \rightarrow \textit{Clingo}$$

Figure 2. Translation from an eFLINT specification-scenario pair to Clingo via the intermediate eFLINT core and state trace abstractions (⇒ = translation, → = dependency).

2 Translation Source Language: eFLINT

Normative Specification Languages in General. Each *(formal) specification language* affords the systematic development of complex artifacts called *specifications*. The rigour of a formal language definitions is useful because it ensures that each specification has a precise, unambiguous meaning. Automated analyses can then reveal or confirm (un)desirable properties of specifications and the concepts they model. For example, a formal specification of a communication protocol can be used to regulate a distributed system; events that deviate from the specification can be ruled out beforehand, monitored, or audited via the inspection of system logs.

Normative specification languages are designed to model norms that regulate social systems, *e.g.*, laws and contracts. These languages formalise notions of *permission*, *prohibition*, and *obligation*, and their tools perform systematic reasoning and case analysis. Languages based on Hohfeld's legal framework [4, 8], such as Symboleo [26, 30] and eFLINT [33, 34] are action-oriented, formalising mutable relations between institutional entities: their *powers* and *duties* to act. The focus on actions affords the specification and control over software systems' interactions with the social world. Such systems are also beholden to norms; *e.g.*, systems processing personal data may be subjected to the GDPR.

The eFLINT Language. Here, we give a brief account of the eFLINT language constructs and how they afford the modelling of (normative) domains of discourse. Throughout, we exemplify concepts with the running example Example 1.

Each specification lays the ontological foundations of the model as a collection of record-type definitions. *E.g.*, `controls` is a record with `controller` and `dataset` fields. Each type is also defined alongside a collection of *clauses*, which *instantiate* these types, identifying particular (relationships between) entities. For example, `controls(agent("Amy"),dataset("X-Rays"))` is an instance of `controls`. The specification models the system by building up a collection of *current* instances. However, instances never occur alone; they are always contextualised by an *attribute*, such as *holds* or *actViol*. Users of the language (statically) agree on how these attributes are interpreted as modalities. For example, instances with *actViol* witness particular cases of *violating actions*, because they were not permitted when they were triggered. Later, we see how the eFLINT semantics formalises such meanings by *inferring* some attributes from others. For example, by definition, instances with *created* also have *enabled*. Users model systems by defining clauses like (`Derived from` ...) and (`Terminates` ...) to capture different normative abstractions. For example, associating `Derived from (Foreach user...))` to the `access` type, the user associates *derived* to `access`(-type) instances. Clauses are evaluated in the context of a concrete *current* state. The states only become concrete when the specification is evaluated alongside a *scenario*. The scenario lays out the instance-attribute pair which gets each successive state

started. Typically, the attribute is *trigger*, such that the instance identifies an action that happens in transition to the next state. *E.g.*, because `access` is marked `Act`, its instances can be triggered. Some attributes have effects that persist over subsequent states. *E.g.*, when triggered, `access` instances *create* new `control` instances, which persist until they are *terminated*.

Thus, eFLINT models complex and stateful cyber-social systems, unambiguously specifies which states are (not) desirable, and lays out the details of concrete scenarios.

3 Translation Target Language: Clingo

Logic- and Answer-Set-Programming. The *logic programming* paradigm operationalises formal logics: logical theories correspond to declarative programs, and the search for proofs of formulae corresponds to program execution and expression evaluation. Different logic programming languages and tools strike different compromises between language expressivity, runtime characteristics, guarantees, and so on. For example, [6] overviews the work and tools around the *Datalog* language, which sees active study and application; consider the PhD thesis of André Pecak in 2024 [23].

Answer-set-programming (or *-solving*) is a form of logic programming with an emphasis on modelling complex problems via rich language features. The name comes from the basis on the *answer-set semantics*, which is also called the *stable model semantics* [13]. This semantics is characterised by attributing a set of *stable models* to each program; each is interpreted as a solution to the search problem. [12] studies the formulation of search problems in these languages.

Clingo is an answer-set solving language and toolset [11]. It is well-known in the answer-set solving community for its maturity and its high performance in solving; *e.g.*, [5] reports on the results of an answer-set competition where various Clingo components outcompeted similar tools.

Features of The Target Fragment of Clingo. Recall Figure 2: we ultimately define the semantics of eFLINT by translating arbitrary eFLINT specifications and scenarios to *Clingo programs*: rule sets in the Clingo language. Thus, eFLINT acquires the *stable model semantics* that underlies Clingo. Here, we explain the features of Clingo that we target with our translation. Section 9 discusses the potential to target other languages (*e.g.*, Soufflé) with many of these features.

We pervasively use Clingo's **general logic programming** features, *e.g.*, these are shared with Datalog. Each rule asserts that the truth of a *consequent* term (or 'conclusion') is implied by the conjunction of listed *antecedent* terms (or 'conditions'). Each rule denotes a set of *concrete rules* for all substitutions of the variables. *E.g.*, `wet(Day) :- rainy(Day), cold(Day)` expresses that 'rainy and cold days are wet'. The `:-` can be omitted from rules with no conditions.

Clingo admits **negated** antecedents, matching `not` *a*, expressing *weak* negation or negation *by default* of *a*: the inability to infer the truth of *a*, *i.e.*, promoting the unprovability of *a* to a proof of ¬*a*. *E.g.*, `wet(Day) :- rainy(Day), not sunny(Day)` expresses that 'rainy but not sunny days are wet'. Languages supporting weak negation must address the absence of a *canonical* (*i.e.*, unique) semantic interpretation for certain syntactically valid rules like `p :- not p`. Many solutions have been explored in the literature. Some languages opt to simply disallow these programs. Answer-set solvers like Clingo generally embrace the lack of a unique interpretation and simply enumerate all the separate *stable models* as alternative answers. Intuitively, this enumerates the possible combinations of assignments to logical variables ('models') that satisfy each program rule. For example, `p :- not p` has zero stable models as none satisfies this contradictory rule.

Clingo's answer-set semantics supports the expression of **(integrity) constraints** over the values of logical variables. These take the form of rules with no explicit consequent, which condition the truth of ⊥: logical inconsistency. For example, `:- not wet(monday)` asserts that 'it is inconsistent to assume that Monday is not wet' or, more intuitively, 'Monday must be wet'. We use such constraints in Sections 4 and 5 to protect the abstractions in our intermediate representations. In each stage, we argue that the constraints of prior stages are satisfied, documenting semantic properties.

Clingo admits (syntactically) **nested terms**. For example, rule `controls(agent("Amy"), dataset(D)) :- dataset(D)` is admissible. In the literature, this feature is also called *function symbols*; reflecting the distinction between the *function symbol* constants of nested terms (*e.g.*, `agent` above) in contrast to *predicate symbols* (*e.g.*, `controls` above); only the latter construct terms that are valuated by the model. Some formalisms emphasise their separation by drawing these symbols from different alphabets, but like eFLINT and Clingo, we do not (*e.g.*, `dataset` above acts as a function *and* predicate symbol). Regardless, the ability to nest terms affords significant modularity in definitions. Consider how `controls(user,dataset)` in Example 1 is independent of the syntactic structure of users.

We exploit Clingo's convenient **tuple terms**, which effectively have the empty function symbol. *E.g.*, `(agent,"Amy")` approximates `agent("Amy")`, but only in the former can `agent` bind variables. Note that Clingo forbids tuple consequents or antecedents, so rule `f(x) :- x` is valid but `(f,x) :- x` is not.

Unfortunately, by supporting nested terms, inference is not guaranteed to **terminate**. This problem has no straightforward solution [28], and we make no attempt to solve it. Instead, we inherit the behaviour of the current eFLINT interpreter and Clingo alike: the evaluation of some programs simply does not terminate. For example, the Clingo interpreter diverges on input `f(f(X)) :- f(X). f(0)`. Intuitively, these programs denote infinitely large models. In Section 6, we address the only threat to the termination of our programs: user-defined eFLINT-derivation and -synchronisation rules.

Like eFLINT, Clingo provides **integer theory**. Integer operators (*e.g.*, `<`, `+`, `-`) and integer and string constants (*e.g.*, `100`, `"Douglas"`) are constant symbols given special treatment;

the operators are concretely denoted in infix position, and their applications are normalised under evaluation, as one is likely to expect. For example, in Clingo and eFLINT, `3 + 1` normalises to `4` and `0 < 1` holds true.

Section 6 relies on (and discusses and demonstrates) features that make *individual* Clingo rules more expressive. Firstly, we use **conditioned conditions** (or 'nested rules') such as (`b : c, d` nested in) `a :- b : c, d ; e`. Thus we model disjunctive conditions; *e.g.*, `x : not y` models $x \vee y$ ($x \leftarrow \neg y$). Note that (`:`) and (`,`) are the nested counterparts of (`:-`) and (`;`), respectively. Secondly, nested rules can be **aggregated** by a preceding `#sum`, `#count`, `#max`, or `#min`, yielding integers. For example, `f(1) :- 0 = #count{x : f(N)}` is contradictory.

Section 7 relies on a feature of Clingo that is atypical outside of answer-set solving: rules with **disjunct consequents**. For example, the Clingo rule `{a}.` *supports* but does not imply that `a` is true. There are two stable models: either nothing is true, or only `a` is true. Section 7 demonstrates how we use these rules to encode search spaces of eFLINT scenarios.

4 State Traces

Essentials of the Event Calculus. eFLINT models dynamic (normative) systems as event-labelled transition systems as was visualised in Figure 1. The original article [34] explicitly bases this approach on the *event calculus*, a logical theory for modelling dynamic systems. It formalises notions of *state*, *fluent*, and their relation as *(holds) in* $\subseteq$ *fluent* $\times$ *state*. The fundamental notion of *inertia* is formalised as an axiom, formalising 'the truth-value of fluents persist from each state to the next until it is acted upon', where actions and states are (inter-)related by a (partial) ordering on *times*. Particular systems are modelled by naming *actions* and specifying their *effects*: how they change truth in the future as a function of truth at present. Particular scenarios are modelled by causing particular actions to happen at particular times.

State Traces. We capture this fundamental layer of the eFLINT semantics in our most abstract intermediate representation: state traces. Each state trace characterises a single eFLINT scenario. We adopt a simplistic model, in which many features of the event and situation calculi coincide: the system is unfolded as a sequence of *states*, indexed by a finite prefix of the natural numbers $\{1, 2, 3, ..., n\}$. We use Clingo's integer theory for traversing and comparing states: $(S+1)$ is the successor of state S, and $(<)$ orders states. We likewise index the sequence of transitions, *i.e.*, each i in $1 \leq i < n$ identifies the ith transition from state i to state $i+1$. We denote the truth fluent F *in* state S as $in(F, S)$; note how we colour these (ubiquitous) *in* terms to aid readability.

Figure 3 lays out the semantics of state traces as a Clingo rule set. The first four rules characterise the states: *state* predicates some non-empty and contiguous prefix of the natural numbers. Rule (INERTIA) models the main axiom of the event calculus: F is true in each state where it was

Notation 1 (Semantic Rules in Clingo)**.** *As per the tradition of formal languages, when they encode our semantics, we notate Clingo rules in the style of the Gentzen-style sequent calculus or natural deduction. For brevity, we fuse rules with identical antecedents by conjoining their consequents.* E.g., *we notate* `a :- f ; g(P,Q)` *and* `b(Q) :- f ; g(P,Q)` *together as* $\frac{f \wedge g(P,Q)}{a \wedge b(Q)}$.

$$\frac{\top}{state(1)} \quad \frac{in(F,S)}{state(S)} \quad \frac{state(S) \wedge 1 < S}{state(S-1)} \quad \frac{state(S) \wedge 0 \not< S}{\bot}$$
$$\frac{in(add(F),S) \wedge \neg in(rem(F),S) \ \wedge \ state(S+1)}{in(F,S+1) \wedge in(add(F),S+1)} \ (\text{INERTIA})$$

Figure 3. Semantics of state traces encoded as Clingo rules, defining the contiguity of state-identifiers in $\{1, 2, 3, ..., n\}$, and encoding the inertia axiom of the event calculus: each $add(F)$ implies the truth of fluent F until after $rem(F)$.

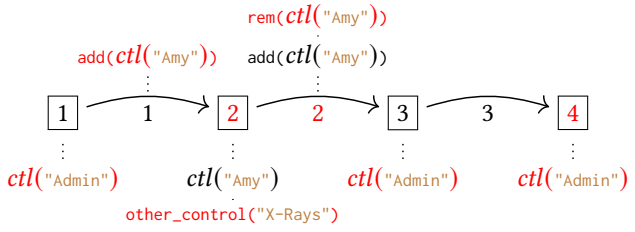
Definition 1 (state-internal)**.** *Rule r is state-internal iff, for each antecedent identical to $in(F, S)$ for some F and S, each consequent of r is identical to $in(F', S)$ for some F'.*

Example 2 (state-internal rules)**.** *Intuitively, state-internal rules do not let truths of the form $in(F, S)$ 'leak' out of S.*

- *Rule* `in(f,S) :- state(S)` *is state-internal because no antecedent matches* `in(F,S)` *for any F and S.*
- *Rule* `in(x,S) :- in(y,S) ; f(y)` *is state-internal because* `in(x,S)` *is identical to* `in(Y,s)` *where $Y =$* `x`.
- *Rule* `in'(x,S) :- in(y,S)` *is not state-internal because constant* `in'` *is not identical to constant* `in`.
- *Rule* `in(x,1) :- in(y,S)` *is not state-internal because constant* `1` *is not identical to variable* `S`.

Example 3 (example Clingo rules defining a state trace)**.** *When the following rules are input to the Clingo interpreter along with the state-trace semantic rules (Figure 3), the output is as visualised below: the states, transitions, and fluents comprising a particular state trace. We colour each term red or black iff it is the conclusion of an example or state-trace semantic rule, respectively. The figure uses $ctl(X)$ as an abbreviation for the Clingo term* `controls(`X`,"X-Rays")`.

```
in(add(controls(user("Amy"),dataset("X-Rays"))),1).
in(rem(controls(user("Amy"),dataset("X-Rays"))),2).
state(3). state(4).
in(dataset(D),S) :- in(controls(A,D),S).
in(other_control(D) ,S) :-
  in(controls(A,D),S), not user("Admin") = A.
in(controls(user("Admin"),D),S) :-
  in(dataset(D),S), not in(other_control(D),S).
```



added (with $add(F)$) more recently than it was removed (with $rem(F)$). Note that this holds even in cases where the same F is added and removed simultaneously, in which case the removal suppresses addition (which results in simpler semantic rules than the alternative). Truths $in(add(X), S)$ and $in(rem(X), S)$ have no direct influence on $in(X, S)$, but rather affect the truth of X in state $S+1$ and onwards. It is helpful to think of $in(add(X), S)$ and $in(rem(X), S)$ as attributes of the *transition* indexed by S, which goes from state S to state $S+1$.

The library defined in Figure 3 affords the definition of state traces using *add* and *rem* in rules to control the presence/absence of fluents in states. The following definition describes a syntactic property over rules that ensures truths about the presence/absence of fluents in one state cannot be used to conclude the presence/absence of fluents in another state; only Rule (INERTIA) is allowed to do this.

Example 3 defines a state trace using only state-internal rules atop the state-trace semantics. The example demonstrates the behaviour of state traces in general, by modelling a facet of the running example in particular (Example 1): Amy sometimes controls the X-Rays dataset, and the administrator controls datasets that have no other controller. Owing to the stable model semantics, the natural language description is captured precisely by these rules and the conditions of the administrator's control are clear: per state, the administrator controls `"X-Rays"` if and only if Amy does not.

5 Core eFLINT

The library defined in this section can simultaneously be seen as a Clingo-embedded domain-specific language (EDSL) and as a core-language underneath eFLINT as one of possibly many surface-languages. In Sections 6 and 7 the library is used as a target language in a translation from eFLINT. As a separately useful EDSL, the library can be used to model dynamically evolving scenarios and reason with the general notions of action- and duty-compliance as defined in [34].

Core eFLINT refines the prior state trace abstraction; as before, *fluents* are related to *states* by the *in* relation. But now each fluent F is an *attribute-instance* pair (A, I), *i.e.*, each $in((A, I), S)$ attributes A to instance I in state S. Instances are opaque in the semantics, *i.e.*, they are always bound to Clingo variables, but instances are further structured by the users (*e.g.*, by specifications in Section 6). Core eFLINT fixes the *attributes* $\mathcal{A}$ in Table 1, and prescribes their semantics in alignment with the corresponding semantics of eFLINT, as it is described [33, 34] and with missing details drawn from the Haskell code of the original interpreter.[2]

Precisely, Figure 4 defines the semantics of Core eFLINT. These rules relate the attributes of each instance. Note that these rules are all state-internal (Definition 1); (only) the **above** rules in Figure 4 use *add* and *rem* to relate the present and future attributes of each instance. Intuitively, the ephemeral truth of *transition*-attributes (in $\mathcal{A}_t$) (like *create*) affect the persistent truth of *state*-attributes (in $\mathcal{A}_s$) (like *created*) and *enabled*. It is helpful to think of $in((A, I), N)$ as labelling the N^{th} transition when A is a transition attribute, *i.e.*, labelling the transition from the N^{th} state to the $N+1^{\text{st}}$ state.

Rules in Figure 4 are clustered into {above, below, right}, formalising three conceptually distinct semantic facets of the original eFLINT. We discuss each in turn.

(When added to Figure 3) the **above** rules in Figure 4 relate fluents with transition attributes *create*, *terminate*, and *obfuscate* to the post-state. For example, $in((create, X), S)$ implies adding $in((created, X), S+1)$. The *obfuscation* of instances is not represented explicitly; it is implied by the absence of *created* and *terminated*. Thus, Core eFLINT captures a form of three-value logic. The conditions on these rules recreate eFLINT's existing, static prioritisation of different effects: termination is prioritised over obfuscation, and creation over obfuscation and termination.

The **below** rules in Figure 4 attribute *holds* and *enabled* to instances, based on whether the instances is created, terminated, obfuscated, derived, or suppressed. The two rules attributing *holds* to instances show the distinct roles of creation and termination; in summary, instance X holds if it is either derived (and not terminated or suppressed) or created, thus abstracting away the details on how the truth of X was established. The *enabled* instances hold while not *suppressed*. Suppression lets eFLINT users express conditions that must always be true for an instance to be enabled, regardless of its other attributes. In practice, this lets (the creators of) eFLINT specifications (which use *suppress*) constrain the enabled actions, despite their lack of control over (the creators of) eFLINT scenarios (which use *create*).

Finally, the **right** rules in Figure 4 define eFLINT's dual notions of normative violation. Duty violations arise in states when instances are enabled while their violations conditions are met. Action violations arise in transitions when action instances are triggered despite being disabled. Note how the *enabled* attribute occurs in both rules; intuitively, it lets agents use the (finite) enabled instances in each state to guide their search for extensions to the scenario that avoid violations. For example, if a duty for Amy is enabled, Amy

$\mathcal{A}_t$	create, terminate, obfuscate, **actTrigger**, actViol, trigger
$\mathcal{A}_s$	**created**, **terminated**, **holds**, **enabled**, dutyViol, **derived**, **suppressed**, **violated**, enum

Table 1. Definition of *(instance) attributes* $a, a_1, a', \ldots : \mathcal{A} \triangleq \mathcal{A}_t \cup \mathcal{A}_s$, associated with states ($\mathcal{A}_s$) and transitions ($\mathcal{A}_t$). Attributes are underlined and **boldfaced** when they occur in antecedents or consequents of Core eFLINT semantic rules in Figure 4; *i.e.*, they are input and output of the library, resp.

[2] We adapt the definitions of `CAll`, `HoldsTrue`, `is_holds`, `is_enabled` and so on, from https://gitlab.com/eflint/haskell-implementation/-/blob/cfe73e9a0ed594e384027ea29529da62dab1fe0f/src/Language/EFLINT/Eval.hs

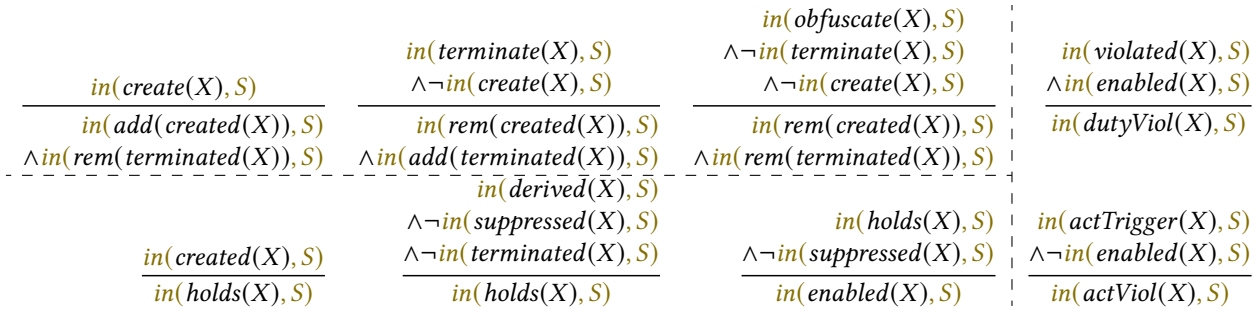


Figure 4. Clingo rules defining the core eFLINT semantics, clustered to reflect three facets of the language. **Above**: the effects of transition attributes in a transition from S to $S+1$ via *add* and *rem*. **Below**: which instances *hold* and are *enabled* as a function of other state-attributes. **Right**: the state- and transition-level attributes which witness normative violations.

should seek to terminate the duty before it is violated. If an action for Amy is enabled, Amy can perform the action (without violation) to achieve certain effects, *e.g.*, terminating a desired duty. In eFLINT today, the search for these desirable actions is externalised, *e.g.*, performed by human users of the eFLINT interpreter. But in Section 7, we internalise this kind of scenario-search problem via Clingo rules.

Example 4 demonstrates a hand-crafted Clingo rule set, defined atop the Core eFLINT semantics, modelling the concrete eFLINT clause `Derived from (Foreach dataset: ... )` in Example 1. This demonstrates how Core eFLINT may be used in its own right. In Section 6 to follow, we define a systematic translation of eFLINT specifications to Clingo rules in this manner, ultimately achieving a similar result.

Example 4 (Default Administrator Control as Clingo Rules)**.**

```
in((derived,non_admin_control(dataset(D))),S) :-
        in((holds,controls(U,dataset(D))),S),
                    not U = user("Admin").
in((derived,controls(user("Admin"),dataset(D))) ,S) :-
       in((holds,                   dataset(D))  ,S),
   not in((holds,non_admin_control(dataset(D)))),S).
```

6 Specification Translation to Core eFLINT

This section defines the eFLINT specification language: its abstract syntax is a formal language $\mathcal{S}$, and its dynamic semantics is given by a translation from $\mathcal{S}$ to Clingo.

6.1 Abstract Syntax

Notation 2 (Common Types and Type-Combinators)**.**

- $\mathbb{Z} \triangleq \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ *is the integer type.*
- $\mathbb{S}$ *is the type of string literals like* `"Well,_howdy_do?"`.
- $A \longrightarrow B$ *is the type of functions from* A *to* B.
- $A \times B \triangleq \{\langle a, b\rangle \mid \forall a{:}A, b{:}B\}$ *is the product of* A *and* B.
- *We use set, map, and list as the usual types of arbitrarily large but finite collections. For example,* $[x, y]$ *is a 2-list. We coerce* $map(A, B)$ *to* $A \longrightarrow B \cup \{\star\}$*, where* $\star \notin B$ *marks each missing mapping, and to or from* $list(A \times B)$*, and we style these pairs* $\langle a, b\rangle$ *suggestively as* $a \mapsto b$*. We coerce* $set(A)$ *to and from* $list(A)$.

$$
\begin{array}{rl}
t, t', \ldots : \mathcal{T} \triangleq \{\mathbb{Z}, \mathbb{S}, \ldots(\text{more identifiers})\} & (\underline{\text{t}}\text{ype ID}) \\
v, v', \ldots : \mathcal{V} \triangleq \mathcal{T} \times \mathbb{S} & (\underline{\text{v}}\text{ariable}) \\
g, g', \ldots : \mathcal{G} \triangleq sum \mid count \mid min \mid max & (\underline{\text{a}}\text{ggregator}) \\
e, e', \ldots : \mathcal{E} \coloneqq struct(t, x : list(\mathcal{E})) & \\
\quad \mid proj(e, v) \mid agg(g, l) \mid var(v) & \\
\quad \mid int(x : \mathbb{Z}) \mid str(x : \mathbb{S}) & (\text{inst. }\underline{\text{e}}\text{xpr.}) \\
p, p', \ldots : \mathcal{P} \subseteq \mathcal{E} \text{ (just } struct, int, str, var) & (\text{inst. }\underline{\text{p}}\text{att.}) \\
i, i', \ldots : \mathcal{I} \subseteq \mathcal{P} \text{ (just } struct, int, str) & (\underline{\text{i}}\text{nstance}) \\
l, l', \ldots : \mathcal{L} \coloneqq for(p, a, l) \mid let(p, e, l) & \\
\quad \mid where(l, b) \mid e & (\text{inst. }\underline{\text{l}}\text{ist}) \\
b, b', \ldots : \mathcal{B} \coloneqq true \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b & \\
\quad \mid e_1 < e_2 \mid e_1 = e_2 \mid check(a, e) & (\text{bool expr.}) \\
c, c', \ldots : \mathcal{C} \coloneqq derive(l) \mid affect(x : \mathcal{A}_t, l) & \\
\quad \mid filter(a, b) \mid actType & \\
\quad \mid infinite \mid finite(x : list(\mathcal{I})) & (\underline{\text{c}}\text{lause}) \\
s, s', \ldots : \mathcal{S} \triangleq map(\mathcal{T}, list(\mathcal{V}) \times set(\mathcal{C})) & (\underline{\text{s}}\text{pec.}) \\
\hline
exists(p, b) \triangleq int(0) < agg(count, for(enum, p, where(p, b))) & \\
forall(p, b) \triangleq \neg exists(p, \neg b) & \\
\hline
fieldPatt(t, s) \triangleq struct(t, fields) \text{ where } \langle fields, x\rangle \triangleq s(t) &
\end{array}
$$

Figure 5. Abstract syntax of eFLINT specifications $\mathcal{S}$, where the attributes $a : \mathcal{A}, \mathcal{A}_t$ are defined in Table 1.

$$
\begin{array}{rr}
type(agg(g, l)) \mid & type(int(x)) \triangleq \mathbb{Z} \\
 & type(str(x)) \triangleq \mathbb{S} \\
type(\langle t, x\rangle : \mathcal{V}) \mid & type(struct(t, x)) \triangleq t \\
type(where(l, b)) \mid & type(for(p, a, l)) \triangleq type(l) \\
 & type(proj(e, v)) \triangleq type(v)
\end{array}
$$

Figure 6. The type of instance (list) expressions.

Figure 5 defines an abstract syntax for eFLINT specifications as its own formal language. It adheres closely to that which was first defined in [34] and then generalised in [33].

Each eFLINT *specification* $s : \mathcal{S}$ consists of mappings from user-defined *(record) types* $t : \mathcal{T}$ to *type definitions* of the form $\langle V, C \rangle$: respectively, the *fields* and *clauses* of t. This definition enables the construction of t-type instances, whose structure is fixed by $V = [v_1, v_2, ..., v_n]$; t acts as a constructor for t-type instances from smaller instances.

Ultimately, Section 6.4 defines the dynamic semantics of each specification in terms of its clauses. Intuitively, each clause of each type t is evaluated in the context of a *current* state S to give t-type instances new attributes in S. The clauses themselves fix the attributes, but the instances arise from the evaluation of instance expressions in $\mathcal{E}$. As a simple example, $derive(struct(instant, [int(9)]))$ gives `instant(9)` the attribute *derived*. The *for* construct generalises these expressions to *lists* of instances, by quantifying over instances with a given attribute, and binding these to variables in a new local scope. The *where* construct filters these lists. For example, $for(enabled, p, where(p, check(dutyViol, p)))$ enumerates the current duty violation instances with the type of instance pattern p. Variables are the simplest patterns, binding quantified instances in their entirety. Other patterns quantify instances and bind their sub-instances to many variables. For example, if $for(struct(controls, [var(\langle agent, \texttt{"X"} \rangle, \langle dataset, \texttt{"Y"} \rangle)]), enabled, l)$ quantifies enabled `controls`-type instances, but binds its controller and dataset to variables named `x` and `y`, respectively. Like *for*, *let* binds instances to variables via patterns, but without any quantification.

6.2 Instance Types and Static Semantics

Only when type t has defined **fields** $F : list(V)$ can any *instance* $i : \mathcal{I}$ *of type* t be constructed. The fields F determine the structure of t-type instances. Instances are constructed with *struct* and de-constructed (projected to a given field) with *proj*, and only when the instances are (de)constructed with the specified fields; *e.g.*, $struct(t, [])$ is well-typed iff t has fields $[]$. We require each types' fields to be distinctly identified (*i.e.*, fields are map-like) such that each projection is unambiguous. Figure 6 defines the (static) typing function, such that evaluating each instance expression $e : \mathcal{E}$ yields an instance $i : \mathcal{I}$ such that $type(e) = type(i)$. Example 5 demonstrates by defining all fields in the running example.

The *primitive types* integers ($\mathbb{Z}$) and strings ($\mathbb{S}$), whose instances cannot be (de)constructed, are implicitly defined. Instead, *int* and *str* introduce their instances as literals in the program text. New integers can also emerge dynamically, from inbuilt operators $\{+, -, \times, \div, \%\}$. These operators are treated as ordinary constants in the abstract syntax, but their special status is reflected by their infix notation in the concrete syntax, and by their reduction under evaluation. For example, users expresss $struct(+, int(2), int(3))$ concretely as $2 + 3$, which is then reduced to $int(5)$ under evaluation.

The static semantics of eFLINT recognises specifications which only use types and variables as one might expect: 1. variables are bound to instances with matching types, 2. per $struct(t, x)$, x binds (exactly) the fields defined for t, 3. projections refer only to existing fields of the given instance (not a literal), and 4. accessed variables occur exactly once in patterns of an enclosing *let*- or *for*-expression.

6.3 Restricted Normal Form

In addition to a specification (and scenario) being well-typed, our translation requires that a specification is in a particular normal form of a restricted $S' \subset S$. The normal form is characterised by the following properties.

(P_1) **Boolean expressions are in conjunctive normal form**. In other words, there is no $\{\neg, \vee, \wedge\}$ inside any $\neg e$, and no $\{\vee, \wedge\}$ inside any $e_1 \vee e_2$. This works around a syntactic limitation of Clingo: antecedents can express $e_1 \vee e_2$ as e_1`: not` e_2, but such terms cannot be nested.

(P_2) **In each** $for(e, a, l)$ **and** $check(a, e)$**,** e **is a** $struct(...)$**.** Ultimately, this ensures that eFLINT's instance types are preserved in Clingo: each term matching `in((A,e),S)` necessarily quantifies instances of $type(e)$. Where $type(e)$ is primitive, the expression is trivial, as primitive instances have no attributes. In the more difficult case, where e is some $var(v)$, we can statically *de-structure* the variable by replacing it with $fieldPatt(type(v), s)$ as defined in Figure 5 (and renaming the new variables to avoid name collisions, if necessary).

(P_3) **There are no projection expressions (with** *proj***).** Because Clingo has no analogue for projection, we rewrite any given $proj(e, v)$. If it is well-typed, $type(e)$ is not primitive, so e is either a variable or a structure. In the first case, we first replace it with $fieldPatt(type(v), s)$, renaming variables as needed. In both cases we get $proj(struct(t, [e_1, e_2, e_3, ...]), v)$, which we rewrite to e_k, where v is the k^{th} field of t.

(P_4) **In each** $agg(g, l)$**, term** l **contains no** $\{agg, \vee\}$**.** Fortunately, Clingo and eFLINT have corresponding syntax and semantics for aggregators. But unfortunately, only eFLINT permits aggregators to be arbitrarily nested. A nested $agg(g, l)$ can be removed via *reification*: it is replaced by the term $struct(t, \langle var(v) \rangle)$, where v is a fresh variable of the fresh type t defined with fields $[\mathbb{Z}]$; intuitively, t collects the elements to be aggregated. In Example 4, `non_admin_control` reifies the nested eFLINT expression (`Exists user: ...`) in Example 1.

(P_5) **Each** agg **is in the** x **or** b **of a** $let(p, e, l)$ **or** $where(l, b)$**.** This works around Clingo's limitations on aggregations in consequents. For example, `f(X) :- X = #count{g(Y)}` is valid Clingo, but `f(#count{g(Y)})` is not. It suffices to replace $agg(g, l)$ with $let(var(v), agg(g, l), var(v))$ for a fresh v, which achieves a similar result; the aggregation becomes an antecedent.

6.4 Dynamic Semantics of eFLINT Specifications

The dynamic semantics of a specification s is defined as a translation to state-internal Clingo rules from the *clauses* $C \subseteq \mathcal{C}$ associated with each type definition $t \mapsto \langle V, C \rangle$ in s. The

Example 5 (specifying fields (not clauses) of Example 1). *For legibility, we use x to abbreviate each variable $\langle x, \texttt{"x"}\rangle$.*

$$
\begin{aligned}
[\quad & \mathit{instant} \mapsto \langle[\mathbb{Z}], []\rangle, \quad \mathit{user} \mapsto \langle[\mathbb{S}], []\rangle,\\
& \mathit{dataset} \mapsto \langle[\mathbb{S}], []\rangle, \quad \mathit{elapsed} \mapsto \langle[\mathit{instant}], []\rangle,\\
& \mathit{controls} \mapsto \langle[\langle \mathit{user}, \texttt{"controller"}\rangle, \mathit{dataset}], []\rangle,\\
& \mathit{access} \mapsto \langle[\mathit{user}, \mathit{dataset}, \mathit{instant}], []\rangle,\\
& \mathit{must\text{-}notify} \mapsto \langle[\mathit{user}, \langle \mathit{user}, \texttt{"controller"}\rangle,\\
& \qquad \mathit{access}, \langle \mathit{instant}, \texttt{"deadline"}\rangle], []\rangle,\\
& \mathit{notify} \mapsto \langle[\mathit{user}, \langle \mathit{user}, \texttt{"controller"}\rangle,\\
& \qquad \mathit{must\text{-}notify}], []\rangle, \quad]
\end{aligned}
$$

$$
\begin{aligned}
[\![s, t, \mathit{derive}(l)]\!] &\triangleq [l, \mathit{derived}]^{\mathcal{L}} \ .\\
[\![s, t, \mathit{affect}(l, a)]\!] &\triangleq [\mathit{for}(p, \mathit{trigger}, l), a]^{\mathcal{L}} \ .\\
[\![s, t, \mathit{filter}(b, a)]\!] &\triangleq [\mathit{for}(p, \mathit{enum}, \mathit{where}(p, b)), a]^{\mathcal{L}} \ .\\
[\![s, t, \mathit{actType}]\!] &\triangleq [\mathit{for}(p, \mathit{trigger}, p), \mathit{actTrigger}] \ .\\
[\![s, t, \mathit{infinite}]\!] &\triangleq [\mathit{for}(p, \mathit{holds}, p), \mathit{enum}] \ .\\
[\![s, t, \mathit{finite}([i_1, i_2, \ldots])]\!] &\triangleq [i_1, \mathit{enum}]^{\mathcal{L}} \ . \ [i_2, \mathit{enum}]^{\mathcal{L}} \ . \ \ldots\\
&\quad \text{where } p \triangleq \mathit{fieldPatt}(t, s)
\end{aligned}
$$

$$
\begin{aligned}
[\mathit{for}(p, a', l), a]^{\mathcal{L}} &\triangleq [\mathit{where}(l, \mathit{check}(a', p)), a]^{\mathcal{L}}\\
[\mathit{where}(l, b), a]^{\mathcal{L}} &\triangleq [l, a]^{\mathcal{L}} \ \texttt{;} \ [b]^{\mathcal{B}}\\
[\mathit{let}(p, e, l), a]^{\mathcal{L}} &\triangleq [l, a]^{\mathcal{L}} \ \texttt{;} \ [p]^{\mathcal{E}} \ \texttt{=} \ [e]^{\mathcal{E}}\\
[e, a]^{\mathcal{L}} &\triangleq \texttt{in((}\ [a]^{\mathcal{A}} \ \texttt{,}\ [e]^{\mathcal{E}} \ \texttt{),S) :-}\ \texttt{state(S)}\\
[\mathit{true}]^{\mathcal{B}} &\triangleq \texttt{\#true}\\
[\mathit{check}(a, e)]^{\mathcal{B}} &\triangleq \texttt{in((}\ [a]^{\mathcal{A}} \ \texttt{,}\ [e]^{\mathcal{E}} \ \texttt{),S)}\\
[b_1 \wedge b_2]^{\mathcal{B}} &\triangleq [b_1]^{\mathcal{B}} \ \texttt{;} \ [b_2]^{\mathcal{B}}\\
[b_1 \vee b_2]^{\mathcal{B}} &\triangleq [b_1]^{\mathcal{B}} \ \texttt{:} \begin{bmatrix} b & \text{if } \exists b, \neg\neg b = \neg b_2\\ \neg b_2 & \text{otherwise}\end{bmatrix}^{\mathcal{B}}\\
[\neg b]^{\mathcal{B}} &\triangleq \texttt{not}\ [b]^{\mathcal{B}}\\
[e_1 < e_2]^{\mathcal{B}} &\triangleq [e_1]^{\mathcal{E}} \ \texttt{<} \ [e_2]^{\mathcal{E}}\\
[e_1 = e_2]^{\mathcal{B}} &\triangleq [e_1]^{\mathcal{E}} \ \texttt{=} \ [e_2]^{\mathcal{E}}\\
[\mathit{struct}(t, [e_1, e_2, e_3, \ldots, e_n])]^{\mathcal{E}}&\\
&\quad \triangleq t\texttt{(}\ [e_1]^{\mathcal{E}} \ \texttt{,}\ [e_2]^{\mathcal{E}} \ \texttt{,}\ [e_3]^{\mathcal{E}} \ \texttt{,}\ \ldots\ [e_n]^{\mathcal{E}} \ \texttt{)}\\
[\mathit{agg}(g, l)]^{\mathcal{E}} &\triangleq [g]^{\mathcal{G}}\ \texttt{\{}\ r([l, \mathit{enum}]^{\mathcal{L}})\ \texttt{\}}\\
&\quad \text{where } r \text{ replaces each } \texttt{;} \text{ with } \texttt{,} \text{ and } \texttt{:-} \text{ with } \texttt{:}\\
[r(x)]^{\mathcal{E}} &\triangleq x \text{ for } r \in \{\mathit{var}, \mathit{int}, \mathit{str}\}\\
[]^{\mathcal{G}} : \mathcal{G} \to \mathbb{S} &\triangleq \{\mathit{count} \mapsto \texttt{\#count}, \mathit{sum} \mapsto \texttt{\#sum}, \ldots\}\\
[]^{\mathcal{A}} : \mathcal{A} \to \mathbb{S} &\triangleq \{\mathit{trigger} \mapsto \texttt{trigger}, \mathit{create} \mapsto \texttt{create}, \ldots\}
\end{aligned}
$$

Figure 7. $[\![s, t, c]\!]$ is the translation to Clingo of the clause c in the definition of type t in (restricted) specification s.

translation for a clause $c \in C$ associated with t in s is defined by the function $[\![s, t, c]\!]$ given in Figure 7. A complete translation applies this function to all clauses in a specification (in any order). The result is well-formed if the specification is of the restricted normal form, *i.e.*, satisfies properties $P_1, \ldots, P_5$, described previously. The translation benefits from the declarative nature of Clingo: re-ordering rules or adding white space does not change their meaning.

The first lines of Figure 7 translate the six kinds of clauses; each ultimately gives new attributes to (new) instances. The cases of the $[\![]\!]$ function definition show how the specialised semantics of these clauses are reconciled. For example, the translation makes explicit the instances which are implicitly quantified by *affect* clauses; each effect arises from some triggered instance. Ultimately, the translation proceeds by inductively unfolding syntactic sub-structures; *e.g.*, $[]^{\mathcal{L}}$ separates rule-consequents from -antecedents, and $[]^{\mathcal{G}}$ maps eFLINT's aggregator syntax to the corresponding Clingo.

Much of the complexity of the translation represents a reconciliation of the semantics of eFLINT and Clingo. For example, in eFLINT the current state is implicit because it remains arbitrary, but we make it explicit in our encoding as the Clingo variable `s`. Some complexities arise from idiosyncrasies of eFLINT and Clingo. For example, Clingo's concrete syntax differs inside and outside of aggregator expressions such as `#sum { ... }`, *e.g.*, rules separate antecedents from consequents with `:-` outside, but with `:` inside.

Example 6 demonstrates our translation of a simple clause from the running example Example 1, which was chosen because it was already in the restricted normal form (see Section 6.3). As with the Clingo rules in Example 4, these rules are state-internal, applying to each state `s` individually, relying on underlying Core eFLINT and state trace semantics to relate fluents within and between (resp.) successive states.

Our Generalisations. Our version of the eFLINT specification language generalises the original in three ways. Firstly, our *let* binders are new. Secondly, we generalise binders in *for* (and *let*) from variables to patterns. Thirdly, our *for* quantifies over instances with any attribute and not just the *enumerables* (with *enum*). In principle, these generalisations enable new usages, but we only use them *within* the translation for the clause-transformations described in Section 6.3, so our experiments use none of these generalisations. But Section 9 discusses our interest in exploring eFLINT variants.

7 Scenario Translation to Core eFLINT.

This section completes the translation by defining the scenario language and translating it to Clingo. We distinguish the two scenario representations, reflecting the two use cases of our new eFLINT interpreter. In either case, we assume we are given an eFLINT specification, which has already been translated into Clingo rules, as presented in Section 6.

Example 6. (*translating* `Violated when Holds(elapsed(deadline))`).

$[\![s, must\_notify, filter(b, violated)]\!]$

where $b \triangleq holds(struct(elapsed, [\langle instant, \texttt{"deadline"}\rangle]))$

$\Rightarrow [for(p, enum, where(p, b)), violated]^{\mathcal{L}}$

where $p \triangleq fieldPatt(must\_notify, s)$

$\Rightarrow [where(where(p, b), check(enum, p)), violated]^{\mathcal{L}}$

$\Rightarrow [where(p, b)), violated]^{\mathcal{L}}$; $[check(enum, p)]^{\mathcal{B}}$

$\Rightarrow [p, violated]^{\mathcal{L}}$; `in(holds(`$[b]^{\mathcal{B}}$`),S)` ; `in(enum(`$[p]^{\mathcal{E}}$`),S)`

$\Rightarrow$
```
[p, violated]^L ; in(holds(elapsed(D)),S) ;
    in(enum(must_notify(U,C,A,D)),S)
```

$\Rightarrow$
```
in(violated([p]^E),S) :- state(S)    ;
    in(holds(elapsed(D)),S)          ;
    in(enum(must_notify(U,C,A,D)),S)

in(violated(must_notify(U,C,A,D)),S) :- state(S) ;
    in(holds(elapsed(D)),S)                     ;
    in(enum(must_notify(U,C,A,D)),S)
```

Example 7 (scenario translated from eFLINT to Clingo).

Abstract eFLINT	*Clingo term*
$\langle create, dataset(\texttt{"X-Rays"})\rangle$	`in(create(dataset("X-Rays")),1)`
$\langle create, controls(user(\texttt{"Amy"}), dataset(\texttt{"X-Rays"}))\rangle$	`in(create(controls(user("Amy"), dataset("X-Rays"))),2)`
$\langle create, user(\texttt{"Bob"})\rangle$	`in(create(user("Bob")),3)`
$\langle create, instant(9)\rangle$	`in(create(instant(9)),4)`
$\langle trigger, access(user(\texttt{"Bob"}), dataset(\texttt{"X-Rays"}), 9)\rangle$	`in(trigger(access(user("Bob"), dataset("X-Rays"),9)),5)`

7.1 Use Case: Check if a Scenario is Compliant

The first use case is scenario-compliance (demonstrated in Example 1) in which the interpreter checks the compliance of a given scenario. Precisely, each scenario is in $list(\mathcal{A}_t \times I)$. Intuitively, each scenario labels each 13th transition with an *effect*: assigning a transition-attribute to a given instance. Example 7 shows the running example scenario before and after translation. Note that the translation is straightforward, and the resulting rules are state-internal.

In the dynamic semantics of a scenario, a concrete 'current state' is modified to its successor state in between the elements of a scenario. Recall that the dynamic semantics of a specification (Section 6) refers to an abstract current state `s` and assigns attributes to instances in `s`. The rules generated from an eFLINT specification, alongside the rules of Core eFLINT (Section 5) and state traces (Section 4), thus produces attribute-assignments for all concrete states generated by the scenario according to the clauses of the specification. It is precisely these attribute-assignments that are interesting output to the eFLINT user. For example, the output (*e.g.*, of Example 1) contains created or terminated facts, triggered actions/events (possibly as the result of a triggered action/event), and violations of actions and duties.

7.2 Use Case: Search for Satisfactory Scenarios

In this novel usage, the user encodes a scenario-search problem: Clingo rules which define 1. the search space of possible scenario actions, *e.g.*, via Clingo's *disjuctive-head* rules, and 2. the satisfaction criterion, *e.g.*, via Clingo's constraint rules.

The search problems expressible via these rules enjoy the same freedoms and limitations as expressing search problems with Clingo in general. For example, the search space and search criterion do not need to be state-internal.

Example 8 shows the Clingo encoding of a search-problem that is of particular interest to eFLINT users and the normative domain in general. Precisely, Example 8 *model-checks* the specification against the proposition 'taking only enabled actions violates no duties', captured by the final rule in the negative, such that stable models encode counterexamples. Each solution details the offending counterexample scenario.

Example 8 (Find Act-Compliant, Non-Compliant Scenarios).

```
1 = { choose(I,S) : triggerable(I,S) } :- state(S).
in((trigger,I),S) :- choose(I,S).
:- 0 = { in((dutyViol,I),S) }.
- - - - - - - - - the case-specific part - - - - - - - - -
{ state(S) } :- S = 1..1000.
triggerable(X,S) :- in((enabled,X),S) ; state(S + 1) ;
       X = ( access(U,D,I) ; notify(U,C,M) ).
in((create,controls(user("Amy"),dataset("X-Rays"))),1).
in((create,controls(user("Bob"),dataset("Biopsy"))),1).
in((create,controls(user("Dan"),dataset("Weight"))),1).
```

The case-specifics define the search space: its depth is limited to 1000 states, and its breadth is defined by the triggerable (`access`- or `notify` action type) instances, which result from the initially created `control`-instances. This input yields no stable model; the property holds! But the scenario is not interesting because no (triggerable) actions advance the `elapsed` time. Adding rule `in((create,elapsed(S)),S) :- state(S)` conflates the passage of states with the passage of time; consequently, counter-examples arise, *e.g.*, where Dan accesses the X-Rays, but violates the duty to inform Amy at instant 11.

8 Correctness Evaluation

We characterise the correctness of our new eFLINT semantics as matching input/output behaviours of the two implementations, where expected. Our artefact includes our source code, experimental data, experimental scripts, and a utility tool for comparing output behaviours (*e.g.*, see Figure 8).

Evaluation Scope. We focus on the existing use case of *checking* given scenarios (see Section 7.1), because our *search* for scenarios is incomparable to in the existing interpreter. For the sake of space, we do not discuss the performance evaluation included in our artefact. Section 9 discusses planned future evaluations, *e.g.*, by comparing our scenario-search to other languages' model-checkers.

Test Suite. Our tests are a suite of specification-scenario pairs chosen to tease out the possible interactions between instances, states, and attributes. Our tests were chosen based

on our understanding of eFLINT, but also, inspired by the exhaustive unit tests in the existing eFLINT interpreter's repository. For example, one test simultaneously creates and terminates an instance, while another spreads these over successive states. The suite also includes the running example, and the *Algemene Afsprakenstelsel* [3], a formalisation on the data sharing agreement of the Dutch Metropolitan Innovations Ecosystem consortium, which is (to our knowledge) the most complex and realistic eFLINT specification at time of writing. All tests, on 108 pairs of scenarios and (43) specifications, are reproducible via our artefact [19]. To follow, we explain the results via a few illustrative test cases.

Evaluation Results. The old and new implementations disagree only in cases of *reasoning by default*, where conclusions are conditioned on the implicit or explicit *absence* of fluents, namely, the evaluation of *where*(*l*, *check*(...)) and *agg*(...), change when fluents are removed. The `Not(Exists...)` expression in Example 1 is such a case. From the user perspective, the existing interpreter can 'overlook' conditions of this form. This emerges from the reasoning algorithm it implements; in summary, it interleavedly evaluates expressions and adds to the collection of fluents, but it does not keep track of which *absent* fluents were checked; later additions can overturn prior checks, so the order of clause-evaluation (which is sensitive to type and variable naming) matters. Example 9 demonstrates with small specifications.

Example 9 (Where the Two eFLINT Semantics Disagree)**.** *For the following two specifications, both versions of the eFLINT interpreter agree that only* `y("")` *holds. Indeed, these specifications have the same stable model, constructed by applying the second derivation clause, after which the first clause is not applicable. Here, we omit* `Identified by String` *for each type.*

```
Fact x Derived from x("") Where Not(Holds(y("")))
Fact y Derived from y("")
```

– –

```
Fact z Derived from z("") Where Not(Exists y: True)
Fact y Derived from y("")
```

Our new interpreter reaches the same result given the following, third version of the specification. But the order in which the existing interpreter evaluates the clauses is unexpectedly disturbed, changing its result. It lets `x("")` *hold, 'forgets' the prior condition on* `y("")`*, and finally lets* `y("")` *also hold.*

```
Fact x Derived from x("") Where Not(Exists y: True)
Fact y Derived from y("")
```

For the same reason, the interpreters' behaviours diverge then the input has no unique stable model. Examples 10 and 11 demonstrate. In either case, users of our new interpreter are better informed. In Example 10, its output makes the unsatisfiability of the specification explicit, rather than silently violating the logical formula encoded in the derivation clause. In Example 11, its output recognises logical 'nondeterminism', enumerating the consistent interpretations of the specification, rather than arbitrarily picking one.

Example 10 (No Stable Model)**.** *Given the following input, our interpreter gives no stable model, which is distinct from one empty stable model. The existing interpreter lets* `p("")` *hold.*

```
Fact p Identified by String
  Derived from p("") Where Not(Holds(p("")))
```

Example 11 (Several Stable Models)**.** *Given the following input, our interpreter gives two stable models: in one,* `f(0)` *holds but* `f(1)` *does not, and vice versa in the other. The existing interpreter agrees with just the second stable model.*

```
Fact f Identified by Int
  Derived from f(0) Where Not(Holds(f(1)))
  Derived from f(1) Where Not(Holds(f(0)))
```

Ultimately, we observe that the two interpreters produce equivalent results, except where they differ as expected, where only the new interpreter reasons by default. We conclude that we have correctly redefined and reimplemented eFLINT, but we continue to expand our test suite.

9 Related & Future Work

Further Testing and Benchmarking. Work continues to expand our test suite. We intend to cover the entirely of the original eFLINT examples, and to benchmark the new interpreter's performance to that of the old interpreter. Much of this performance evaluation has already been done and can be found in the artefact [19], but we have left it out of scope. So far, we find that the new interpreter dramatically outperforms the original in most cases (reaching 25× speedup in one long and complex scenario), but their performance is comparable in trivial cases, or in cases when Clingo's optimisations fail, *e.g.*, in guessing prime numbers.

Different Ways to Define eFLINT. The impetus for our work was the observation that, *in abstracto*, eFLINT specifications resemble logic programs. Thus, we give eFLINT a semantics via a translation to logic programs. But we acknowledge that other approaches are available also. For example, [1] formalises another language which is also normative and used for case analysis and model-checking. But their formalism is based on first-order linear temporal logic. Consequently, their work is more self-contained; *e.g.*, readers of [1] are not burdened with the idiosyncrasies of Clingo. But our approach has the benefit of leveraging the existing Clingo interpreter. If the trade-off changes in the future, newer semantics can always be defined and related to this one, *e.g.*, as was done extensively for the Reo language [14].

eFLINT Simplifications & Variants. Our translation faithfully captures semantic facets of eFLINT in attributes. However, some facets could be captured in user-defined instances and types instead. For example, if termination is

[3]The document is available at https://dmi-ecosysteem.nl/wp-content/uploads/bb_documents/2023/10/2023.06.01-DMI-Afsprakenstelsel-v1.pdf, our formalisation process is documented at https://definities.dmi-ecosysteem.nl, and both are alongside our eFLINT formalisation in our artefact [19].

removed, users can model the termination of i by creating i' if i' suppresses i. And if each *finite*($[i_1, i_2, ..., i_n]$) clause is approximated by n by *derive* clauses, *emit* can be removed, because its instances coincide with those that *hold*. Thus, simpler specifications become easier to model and reason about. We also intend to experiment with Core eFLINT. Which semantic changes are desirable? Can we create a common core for eFLINT and related languages such as DPCL [31]?

Circumstantial Substitutes for Clingo. In this work, we prioritised the cohesion in our semantics accross the two use-cases of eFLINT. But in the simpler use case of scenario-checking, we make little use of Clingo's more powerful features. In the future, we are interested in specialising the translation pipeline to target languages more suited to this specialised task. For example, we consider instead targeting the Datalog-based *Soufflé* language [15]. Unlike Clingo, Soufflé has eFLINT-like record types and type checking. Can we translate between these notions of type? Does this result in more efficient reasoning and more explainable output?

Logical Theories of Dynamic Systems. Like the original eFLINT article [34], in Section 4, our semantics draws from work on the event-, situation-, and fluent-calculi. Other works (*e.g.*, [27]) thoroughly overview and compare these formalisms. Here, it suffices to acknowledge that our notions of state and action were originally inspired by the event calculus [29], but in our simple case of linear traces, many features of the other calculi emerge also. Like *situations*, our scenarios list actions in the order they happen [17, 18].

Many-Valued Logics. Prior works have generalised the usual Boolean value domain of logical variables [3, 21, 22]. For example, [21] distinguishes *essential* from *accidental* truth, and distinguishes their semantics under combination with $\wedge$, $\neg$, and so on, to capture 'natural language reasoning'. Regardless, the system can be reasoned about; *e.g.*, [20] axiomatises a first-order fragment of the system, and proves its completeness. Similarly, we distinguish *attributes* of the same instances to encode many truth values, and our semantic rules define how they interact. Also similarly, we prescribe distinct (modal) interpretations of different values; *e.g.*, *actViol* truths are undesirable, because they represent actions taken without permission. We also model a three-valued logic in Core eFLINT in (interactions between) *create*, *terminate*, and *obfuscate*, to encode priority.

Constraint- vs. Logic Programming. An experimental eFLINT model-checking backend was developed in 2022 [7], inspired by the approach of Symboleo: targeting the nuXmv SMT solver to model-check scenarios. This work inspired our own, but we opted to target Clingo instead of nuXmv, because its stable model semantics aligns more closely with what was originally described for eFLINT. It concerns the distinction between logic- and constraint-programming: only the former requires that truths be *supported* by concrete inference steps, *i.e.*, each truth in the model is the root of a proof tree. In practice, truth is more explainable; *e.g.*, nuXmv but not Clingo lets `x` be true in the trivial specification; Clingo cannot support this truth, as it is the consequence of no rule.

Evaluating Scenario Search. We have left the evaluation of our interpreter's scenario-searching out of scope. In future, we want to more precisely characterise what expressive power this brings to users, and how it compares to tools for other languages such as Symboleo [25] and Fievel [36].

10 Conclusion

In this work, we re-defined the eFLINT normative specification language via a translation to Clingo answer-set solving. We defined a new eFLINT semantics and interpreter.

Our semantics is desirable for its rigour. Notably, we have formalised a notion expressed in the original article [34]: eFLINT clauses express logical inference rules. In the process, we have identified and corrected a flaw in the existing eFLINT interpreter concerning inference about conditions *by default*, which is well-studied in the logic programming community. Our interpreter reasons by default correctly.

In fact, our interpreter consists almost entirely of the Clingo solver; by translating eFLINT to Clingo, we let Clingo perform our normative case analysis. This approach minimises the gap between the semantics and the interpreter. Moreover, it afforded the generalisation of eFLINT's typical scenario case analysis to scenario *search*. Via Clingo, our tool can recommend scenarios and model-check scenarios against properties in general, and specification-compliance in particular. These use cases were supported in languages related to eFLINT, but not yet in eFLINT itself.

In the future, we want to continue improving and evaluating our new interpreter, for example, by drawing from ruleset preprocessing from the answer-set solving community. We are also interested in investigating the intermediate languages we have identified between eFLINT and Clingo. Our *Core eFLINT* language is promising: its embedding in Clingo makes it inter-operable with any Clingo rules, and it includes much of the eFLINT semantics, but it more abstract and simple. In the future, we want to investigate these kinds of minimal normative languages as new translation targets for various purposes, and for their suitability – in their own right – to the use cases of eFLINT: normative specification, automated case analysis, and normative model-checking.

Data Availability Statement. Our artefact [19] includes the source code of our implementation, and the data and scripts necessary to (re)produce our evaluation in Section 8.

Acknowledgements. This research is partially funded by the AMdEX-fieldlab project (Kansen Voor West EFRO grant KVW00309) and the AMdEX-DMI project (Dutch Metropolitan Innovations ecosystem for smart and sustainable cities, made possible by the Nationaal Groeifonds).

References

[1] Huib Aldewereld, Javier Vázquez-Salceda, Frank Dignum, and John-Jules Ch. Meyer. 2005. Verifying Norm Compliancy of Protocols. In *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems, AAMAS 2005 International Workshops on Agents, Norms and Institutions for Regulated Multi-Agent Systems, ANIREM 2005, and Organizations in Multi-Agent Systems, OOOP 2005, Utrecht, The Netherlands, July 25-26, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3913)*, Olivier Boissier, Julian A. Padget, Virginia Dignum, Gabriela Lindemann, Eric T. Matson, Sascha Ossowski, Jaime Simão Sichman, and Javier Vázquez-Salceda (Eds.). Springer, 231–245. doi:10.1007/11775331_16

[2] Jamila Alsayed Kassem, Corinne Allaart, Saba Amiri, Milen Kebede, Tim Müller, Rosanne Turner, Adam Belloum, L. Thomas van Binsbergen, Peter Grunwald, Aart van Halteren, Paola Grosso, Cees de Laat, and Sander Klous. 2024. Building a Digital Health Twin for Personalized Intervention: The EPI Project. In *Commit2Data (Open Access Series in Informatics (OASIcs), Vol. 124)*, Boudewijn R. Haverkort, Aldert de Jongste, Pieter van Kuilenburg, and Ruben D. Vromans (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:18. doi:10.4230/OASIcs.Commit2Data.2

[3] João Barbosa, Mário Florido, and Vítor Santos Costa. 2019. A Three-Valued Semantics for Typed Logic Programming. In *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019 (EPTCS, Vol. 306)*, Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Inclezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang (Eds.). 36–51. doi:10.4204/EPTCS.306.10

[4] Pierfrancesco Biasetti. 2015. Hohfeldian normative systems. *Philosophia* 43 (2015), 951–959.

[5] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. 2016. Design and results of the Fifth Answer Set Programming Competition. *Artif. Intell.* 231 (2016), 151–181. doi:10.1016/J.ARTINT.2015.09.008

[6] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (1989), 146–166. doi:10.1109/69.43410

[7] Florine de Geus. 2022. Model Checking Normative Systems. *University of Amsterdam* (2022).

[8] Luís Duarte d'Almeida. 2016. Fundamental legal concepts: the Hohfeldian framework. *Philosophy Compass* 11, 10 (2016), 554–569.

[9] Christopher A Esterhuyse and L Thomas van Binsbergen. 2024. Cooperative Specification via Composition Control. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*. 2–15.

[10] European Commission. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance). https://eur-lex.europa.eu/eli/reg/2016/679/oj

[11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19, 1 (2019), 27–82. doi:10.1017/S1471068418000054

[12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2022. *Answer set solving in practice.* Springer Nature.

[13] Michael Gelfond and Vladimir Lifschitz. 1988. The stable model semantics for logic programming.. In *ICLP/SLP*, Vol. 88. Cambridge, MA, 1070–1080.

[14] Sung-Shik TQ Jongmans and Farhad Arbab. 2012. Overview of Thirty Semantic Formalisms for Reo. *Scientific Annals of Computer Science* 22, 1 (2012).

[15] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 422–430. doi:10.1007/978-3-319-41540-6_23

[16] Lu-Chi Liu, Giovanni Sileno, and Tom M. van Engers. 2020. Digital Enforceable Contracts (DEC): Making Smart Contracts Smarter. In *Legal Knowledge and Information Systems - JURIX 2020: The Thirty-third Annual Conference, Brno, Czech Republic, December 9-11, 2020 (Frontiers in Artificial Intelligence and Applications, Vol. 334)*, Serena Villata, Jakub Harasta, and Petr Kremen (Eds.). IOS Press, 235–238. doi:10.3233/FAIA200872

[17] John McCarthy. 1963. Situations, actions, and causal laws.

[18] John McCarthy. 2002. Actions and other events in situation calculus. In *Kr.* 615–628.

[19] Tim Müller, Christopher A. Esterhuyse, and L. Thomas van Binsbergen. 2025. Source Code and Experimental Data for a Translatfrom eFLINT Normative Specifications to Clingo Answer-Set Programs. doi:10.5281/zenodo.15188960 This URL and DOI refer to the most recent version of our artefact, from where the URL and DOI of each (fixed) version is accessible. Version1 has DOI 10.5281/zenodo.15188959, and version2 has DOI 10.5281/zenodo.15470286. The latter is the most recent at time of writing and accompanies this article.

[20] Grigory K. Olkhovikov. 2016. A Complete, Correct, and Independent Axiomatization of the First-Order Fragment of a Three-Valued Paraconsistent Logic. *FLAP* 3, 3 (2016), 335–340. http://www.collegepublications.co.uk/downloads/ifcolog00007.pdf

[21] Grigory K. Olkhovikov. 2016. On a New Three-Valued Paraconsistent Logic. *FLAP* 3, 3 (2016), 317–334. http://www.collegepublications.co.uk/downloads/ifcolog00007.pdf

[22] Mauricio Osorio and Claudia Zepeda. 2019. Three New Genuine Five-valued Logics Intended to Model Non-trivial Concepts. In *Selected Papers of the Eleventh and Twelfth Latin American Workshop on Logic/Languages, Algorithms and New Methods of Reasoning, LANMR 2018, Puebla, Mexico, November 15, 2018 & LANMR 2019, Puebla, Mexico, November 15, 2019 (Electronic Notes in Theoretical Computer Science, Vol. 354)*, Pilar Pozos Parra and José Raymundo Marcial-Romero (Eds.). Elsevier, 157–170. doi:10.1016/J.ENTCS.2020.10.012

[23] André Pacak. 2024. *Datalog as a (non-logic) Programming Language.* Ph. D. Dissertation. University of Mainz, Germany. https://openscience.ub.uni-mainz.de/handle/20.500.12030/10930

[24] Mostafa Mohajeri Parizi, L. Thomas van Binsbergen, Giovanni Sileno, and Tom van Engers. 2022. A Modular Architecture for Integrating Normative Advisors in MAS. In *Multi-Agent Systems*, Dorothea Baumeister and Jörg Rothe (Eds.). Springer International Publishing, Cham, 312–329.

[25] Alireza Parvizimosaed, Marco Roveri, Aidin Rasti, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. 2022. Model-checking legal contracts with SymboleoPC. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems* (Montreal, Quebec, Canada) *(MODELS '22)*. Association for Computing Machinery, New York, NY, USA, 278–288. doi:10.1145/3550355.3552449

[26] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, Marco Roveri, Aidin Rasti, Ali Roudak, and John Mylopoulos. 2022. Specification and analysis of legal contracts with Symboleo. *Software and Systems Modeling* 21, 6 (2022), 2395–2427.

[27] Stephan Schiffel and Michael Thielscher. 2006. Reconciling Situation Calculus and Fluent Calculus. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*. AAAI Press, 287–292. http://www.aaai.org/Library/AAAI/2006/aaai06-046.php

[28] Danny De Schreye and Stefaan Decorte. 1994. Termination of Logic Programs: The Never-Ending Story. *J. Log. Program.* 19/20 (1994), 199–260. doi:10.1016/0743-1066(94)90027-2

[29] Murray Shanahan. 2001. The event calculus explained. In *Artificial intelligence today: Recent trends and developments*. Springer, 409–430.

[30] Sepehr Sharifi, Alireza Parvizimosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. 2020. Symboleo: Towards a specification language for legal contracts. In *2020 IEEE 28th international requirements engineering conference (RE)*. IEEE, 364–369.

[31] Giovanni Sileno, Thomas van Binsbergen, Matteo Pascucci, and Tom van Engers. 2022. DPCL: a language template for normative specifications. *arXiv preprint arXiv:2201.04477* (2022). doi:10.48550/arXiv.2201.04477 Available at https://arxiv.org/abs/2201.04477..

[32] Jorrit Stutterheim, Aleandro Mifsud, and Ana Oprescu. 2024. DYNAMOS: Dynamic Microservice Composition for Data-Exchange Systems, Lessons Learned. In *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*. 8–15. doi:10.1109/ICSA-C63560.2024.00008

[33] L. Thomas van Binsbergen, Milen G. Kebede, Joshua Baugh, Tom M. van Engers, and Dannis G. van Vuurden. 2021. Dynamic generation of access control policies from social policies. In *The 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2021) / The 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2021), Leuven, Belgium, November 1-4, 2021 (Procedia Computer Science, Vol. 198)*, Nuno Varandas, Ansar-Ul-Haque Yasar, Haroon Malik, and Stéphane Galland (Eds.). Elsevier, 140–147. doi:10.1016/J.PROCS.2021.12.221

[34] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom M. van Engers. 2020. eFLINT: a domain-specific language for executable norm specifications. In *GPCE '20: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Virtual Event, USA, November 16-17, 2020*, Martin Erwig and Jeff Gray (Eds.). ACM, 124–136. doi:10.1145/3425898.3426958

[35] L. Thomas van Binsbergen, Merrick Oost-Rosengren, Hayo Schreijer, Freek Dijkstra, and Taco van Dijk. 2024. *AMdEX Reference Architecture – version 1.0.0.* doi:10.5281/zenodo.10565915

[36] Francesco Viganò and Marco Colombetti. 2008. Model Checking Norms and Sanctions in Institutions. In *Coordination, Organizations, Institutions, and Norms in Agent Systems III*, Jaime Simão Sichman, Julian Padget, Sascha Ossowski, and Pablo Noriega (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 316–329.

[37] Tomasz Zurek, Mostafa Mohajeriparizi, Jonathan Kwik, and Tom M. van Engers. 2022. Can a Military Autonomous Device Follow International Humanitarian Law?. In *Legal Knowledge and Information Systems - JURIX 2022: The Thirty-fifth Annual Conference, Saarbrücken, Germany, 14-16 December 2022 (Frontiers in Artificial Intelligence and Applications, Vol. 362)*, Enrico Francesconi, Georg Borges, and Christoph Sorge (Eds.). IOS Press, 273–278. doi:10.3233/FAIA220479

Experimental Output Visualiser

```
Running diff test ./tests/clingo/diff_4_exists_forall.diff0000.eflint...
1 +-eFLINT---------------+-Clingo---------------+
  | (holds, meta(""))    | (holds, meta(""))    |
  | (holds, no_dubs("")) | (holds, no_dubs("")) |
2 +-eFLINT---------------+-Clingo---------------+
< | (holds, meta(""))    |                      |
  | (holds, no_dubs("")) | (holds, no_dubs("")) |
  | (holds, some(""))    | (holds, some(""))    |
  | (holds, x(2))        | (holds, x(2))        |
3 +-eFLINT---------------+-Clingo---------------+
< | (holds, meta(""))    |                      |
  | (holds, some(""))    | (holds, some(""))    |
  | (holds, x(2))        | (holds, x(2))        |
  | (holds, x(3))        | (holds, x(3))        |
4 +-eFLINT---------------+-Clingo---------------+
< | (holds, meta(""))    |                      |
  | (holds, no_dubs("")) | (holds, no_dubs("")) |
  | (holds, some(""))    | (holds, some(""))    |
  | (holds, x(3))        | (holds, x(3))        |
5 +-eFLINT---------------+-Clingo---------------+
  | (holds, meta(""))    | (holds, meta(""))    |
  | (holds, no_dubs("")) | (holds, no_dubs("")) |
  +----------------------+----------------------+
 > Test OK
1 test(s) pass, 0 test(s) fail, 0 performance test(s)
```

Figure 8. A screenshot of our prototype command-line visualiser tool, comparing the outputs of our new eFLINT implementation (right) against the original (left), for the same specification-scenario input pair. This shows the result of some test input (available in the artefact) where three fluents are inferred by only the original interpreter. This test passes because the difference is expected in this case.