Towards a standardised and efficient implementation of the normative specification language eFLINT

Olaf Erkemeij
o.a.erkemeij@uva.nl
University of Amsterdam
Amsterdam, The Netherlands

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

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

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

ABSTRACT

Companies, organisations, and the software services they provide are under increased scrutiny regarding the (privacy) laws, regulations, and contracts that govern them. Effort is spent to ensure compliance as part of the initial software construction process as well as by auditing software after its construction. To date, these efforts are largely manual and are often opaque to users of the software. The situation can be improved by automating compliance using explicit specifications of legal requirements that formalise interpretations of legal documents. The introduced separation of concerns not only makes it easier to adapt to changes but can also improve transparency when the applied interpretations are made available.

The eFLINT language has been designed as a domain-specific specification language for these purposes. The language has been applied within several research projects and demonstrated the benefits of automating compliance using a domain-specific language and a dedicated reasoner. The language has evolved towards increased flexibility and modularity in order to support a wide range of applications. This paper presents the latest version of the language and discusses implementation strategies for developing an efficient reasoner. A particular challenge is provided by logical derivation rules with negative conditions. Alternative derivation semantics are discussed and evaluated within context. The paper also presents an API for interacting with the reasoner. As such, this paper marks an important step towards a standardised and efficient implementation of the eFLINT language.

CCS CONCEPTS

• **Computer systems organization** → *Architectures*; • **Theory of computation** → *Automated reasoning*; *Operational semantics*; • **Software and its engineering** → *Software performance.*

KEYWORDS

compliance, legal requirements, domain-specific language, language design, interpretation, logic programming, architecture, concurrency

ACM Reference Format:
Olaf Erkemeij, Christopher A. Esterhuyse, Tim Müller, and L. Thomas van Binsbergen. 2018. Towards a standardised and efficient implementation of the normative specification language eFLINT. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Companies and governmental organisations are obligated to adhere to laws, regulations, policies, and other sources of norms in the services they provide. An important question faced by organisations is whether their implementation of services is *compliant* with the relevant norms. Determining compliance is a complex and potentially costly task, especially when both the service and the relevant norms are subjected to (frequent) changes. A possible solution is to (partially) automate compliance through establishing a direct and formal connection between the service implementation and a formal specification of norms. Formal specifications of norms can be directly embedded in software systems for runtime monitoring or decision making and can be used to generate software components that are compliant by construction. Through separation of concerns, the software is easier to adapt to changes in the relevant norms, especially when the normative specifications are modular. Additional benefits are the potential for improved traceability of compliance decisions and transparency for affected stakeholders.

The eFLINT language was designed as a *Domain-Specific Language* (DSL) to formalize norms as executable specifications with the aforementioned benefits in mind [16]. The paper that introduced eFLINT established a formal connection between legal/normative concepts such as powers, duties, and violations and computational concepts such as states and transitions in transition systems. In particular, eFLINT takes an action-based approach, deriving normative positions from actions available to actors, actions actors should perform (duties) and the normative consequences (powers) of actions taken by actors.

In a follow-up paper, language extensions have been proposed that improve the modularity of specifications and make it possible to connect specifications at different levels of abstractions [15]. In particular, the paper shows how system-level access control actions 'inherit' pre- and post-conditions from higher-level specifications such as a data sharing agreement and the GDPR privacy regulation. The language has been applied successfully in various contexts such as multi-agent, normative systems [10] and data exchange systems [2]. Within the AMdEX project, the language is applied within real-world use cases provided by public and private partners. In the process, the language underwent several modifications and extensions and no complete (formal) description of the semantics of the language has been provided. Moreover, the interpreter used in the aforementioned experiments qualifies as experimental research software, and has not yet been standardised or optimised for performance. Of particular relevance are the semantics of the logical derivation rules in eFLINT, their interaction with the effects of actions, and the handling of negative antecedents. These aspect provide particular challenges to efficient implementation.

In this paper we present a more thorough account of the semantics of the language alongside a novel implementation intended to improve efficiency and standardise interactions with the interpreter for various applications. In the process, several design decisions regarding the semantics of the language are discussed, as well as implementation strategies for improved performance.

This paper makes the following contributions:

- A detailed description of the derivation semantics for eFLINT's derivation rules with negative antecedents alongside an algorithm that realises the semantics. The resulting semantics finds a pragmatically motivated point in the design space closely related to the existing stable model semantics and well-founded semantics.
- A discussion on the challenges and opportunities regarding the efficient implementation of the semantics of eFLINT, considering execution time and memory consumption, intended to simultaneously give a detailed account of the semantics of eFLINT and to report on insights relevant to the implementation of alternative interpreters (possibly for similar languages).
- An API specification for a server that standardises the communication with eFLINT-like interpreters of normative specifications (normative reasoners). Servers that adhere to the specification can be used in various kinds of applications, can serve concurrent requests from various clients, and can be used in stateful or stateless manner.

The paper is structured as follows. Section 2 introduces the main ideas behind eFLINT and the general problem of negated conditions in logic programming. Section 3 gives a detailed account of the different constructs of the eFLINT language. Section 4 gives a detailed description of possible derivation semantics for eFLINT. The final version of the paper will qualitatively evaluate the alternatives. Section 5 discusses the server-client architecture of a new implementation of the language, involving an API intended to standardise communication with an underlying interpreter for the language. Section 6 discusses the implementation of the underlying interpreter and evaluates it in comparison to a reference interpreter for the language. The final version of the paper will discuss additional implementation strategies to optimise for performance and contain additional empirical experiments.

2 BACKGROUND

This section introduces the eFLINT language by giving a high-level description of its semantics and the most important semantic approaches to derivation rules with negative antecedents are summarised.

2.1 The eFLINT Language

The eFLINT language is a domain-specific language for the formalisation of norms as normative specifications [16]. The language is intended to be used in a wide-range of applications and application areas in which automated compliance is desired. As such, the language can be used to connect legal concepts to computational concepts and apply existing techniques for reasoning about and controlling system behaviour such as formal verification, run-time verification and access control [15]. To support the formalisation of a wide-range of normative[1] sources, eFLINT is based on fundamental normative concepts derived from Hohfeld's legal framework [4]. The language uses a relational model to capture knowledge that serves as the basis of reasoning about the *normative positions* of actors, i.e., whether actors have certain *permissions*, *powers*, and *duties*. A knowledge base captures normative positions as well as facts that form a model of the state of system being governed by the norms. Examples of facts are whether a permit has been granted or a certain document has been signed. Derivations rules are used to derive knowledge from basic facts stated as facts within a specification (or *postulated* externally, as explained in Section 3). In this sense, eFLINT is comparable to logical programming languages such as Prolog [20] and Datalog, and modelling languages such as Alloy [6].

Languages based on Hohfeld's framework, such as Symboleo [12] and eFLINT, are oriented towards the actions that actors can perform and the effects these actions have. This is in contrast to Deontic logics, which focus on modalities regarding a current state. Most notably, the concept of power is often missing, directly coupling performed actions to changes in normative positions. For example, actor A has a power when A can grant a construction-permit to B such that B acquires the permission to build on a certain location. The actions of eFLINT can be seen to describe functions, mapping an input knowledge base to an effect: a statement about the truth (or falsity) of certain facts. In this sense the language can be seen as functional. However, to reason about a scenario, containing sequences of actions, the effects of actions need to manifest themselves in order to update the normative positions held by actors as well as any other facts. The triggering of actions can thus be seen as an imperative construct, modifying the knowledge base as a global, mutable state. The violations 'raised' as a scenario unfolds can be seen as (additional) side-effects. A central challenge in the (correct and efficient) implementation of eFLINT concerns the

[1]We use the term 'normative' in this paper as a generalisation of 'legal', including besides laws, regulations and contracts, also organisational policies and other descriptions of (un)desirable behaviour.

interplay between the logical derivation rules and the imperative effects (mutations) of actions and events.

The original paper defines a specification as a collection of declarations of fact-, duty-, event- and act-types. The type declarations determine the structure of the knowledge bases admitted by the specification and thereby the structure of the states of a transition system. The event- and act-types (additionally) describe the transitions of the transition system, with events and actions (instances of said types), labelling the transitions. A scenario then describes a trace in the transition system and the pre-conditions of act-types and the violation conditions of duty-types determine which transitions in a trace give rise to violations (if any), i.e. which actions or events cause non-compliance in a certain state. A trace is considered 'action-compliant' if every action labelling a transition in the trace is 'enabled' in the knowledge base corresponding to the source of transition. Similarly, a trace is considered 'duty-compliant' if every state is labelled with a knowledge base in which every duty is not violated according to any of the violation conditions of the corresponding duty-type.

This high-level semantic description of eFLINT is shared by all versions of the language (although the meaning of 'enabled' has undergone some subtle changes). Since the original paper, the language has been extended and modified, primarily to support additional application contexts. The reference interpreter [14], implemented in Haskell and provided along the original publication on eFLINT [16], has been updated accordingly. Most notably, the language has been extended to allow for modular specifications and for the incremental development of specifications and scenarios [17] (through the introduction of *phrases*, see Section 3). In addition, the language was extended to allow for incremental development of individual types (with the Extend keyword) and to allow for actions and events to synchronise [15]. The latter extension makes it possible to explicitly connect normative specifications that regulate the same system at a different regulatory level, such as the GDPR specification and a data sharing agreement (see [15] for details).

Section 3 gives a detailed account of the current version of the language and Section 4. In particular, previous works applied a version of eFLINT without negative conditions in derivation rules. Section 4 compares alternative semantics for derivation rules with negative conditions based on the existing approaches discussed next.

2.2 Derivation Semantics

Logic programming [7, 8], rooted in mathematical logic, is a programming paradigm in which programmers describe the structure of relations as well as propositional formulae that determine membership of relations. These formulae are written as (derivation) rules adhering to prescribed rule formats depending on the chosen semantics and/or implementation. Rules typically encode logical implications (written from right-to-left) such as $p \leftarrow a \wedge b$. The above is an example of a *Horn clause* [5, 18]. A Horn clause is a clause that adheres to the rule format in which a single positive proposition (p in the example) is conditioned by zero or more positive antecedents (a and b in the example). Horn-clauses are suitable for logic programming as they are expressive and can be implemented efficiently. The expressivity of derivation rules significantly increases, however, when negative propositions are accepted as conditions. As an example, consider Figure 1. Negative conditions raise the question how it can be determined that a proposition holds false. A significant body of literature is devoted to this topic.

1. $p \leftarrow \neg q$
2. $q \leftarrow \quad p$
3. $a \leftarrow \neg b$

Figure 1: A set of derivation rules with negative conditions and a circular dependency.

Negation-as-Failure. Negation-as-failure [1] is a fundamental concept in logic programming that posits that the failure to prove a proposition proves its negation. Negation-as-failure has been adopted as an operand in the standard of Prolog [20]. However, negation-as-failure has a procedural aspect (i.e., the algorithmic process attempting to prove a proposition) and does not equal logical negation.

Applying negation-as-failure in the rules of Figure 1, both `p` and `a` can be added to the derived facts, as the values of both `q` and `b` are unknown. This results in the knowledge base `{p, a}`. However, using this knowledge base, the statement `q` can also be added, as `p` holds true. The final result `{p, q, a}` was validly obtained, but contains *inconsistent* results; p was derived on the condition of $\neg q$, which is contradicted by the derived q. This is a problem that occurs with negation of literals in logic programming when imperatively evaluating the rules. To remedy this, different semantics for dealing with negation in derivation rules have been developed over the years. Among them are the `Stable Model Semantics` and the `Well-founded Semantics`. Both models are defined for logic programs that implement negation as failure.

Stable model semantics. The stable model semantics [3] is a declarative semantics that aims to provide a coherent meaning to logic programs that include negation as failure. A stable model represents a consistent and minimal interpretation of a logic program. Concretely, such that (1) all rules hold when evaluated as logical implications, and (2) the conditions of rules deriving facts are consistent with the results.

The stable-model semantics is not *canonical*; it does not determine a unique model per program. A logic program can have zero, one, or many stable models. When there is no stable model, it indicates that the logic program can never be consistent, regardless of the knowledge base. When a logic program has multiple stable models, all solutions are equally valid. For example, an implementation returns whichever is found first, based on the algorithm or the order of the clauses within the program.

Well-founded semantics. The well-founded semantics is another approach to handling negation in logic programming, which was first introduced in 1991 [19]. Most significantly, unlike the stable-model semantics, it is canonical, identifying exactly one model per program. To preserve consistency, the well-founded semantics introduces a third truth value: *unknown*. As such, this semantics has also been presented as a three-value logic [11]. Intuitively, it determines the minimal set of unknown facts such that the knowledge base is consistent. For example, given 1, *a* is true, *b* is false, and *p* and *q* are unknown.

3 EFLINT LANGUAGE OVERVIEW

In this section, an overview of the eFLINT language will be given through the use of a running example. In this example, an election will be modelled, in which citizens can vote on a set of candidates. More documentation and examples can be found on the eFLINT GitLab [13].

At the top level of the eFLINT language are phrases. Phrases can be seen as statements, each of which is executed in sequence. These phrases contain the core functionality of the language, as they directly express what operations the user can perform.

Fact-type declarations. One of the essential phrases within the language is the `Fact` phrase, which is used to define new facts. To define what a citizen is, we can use either of the `Fact` definitions seen in Listing 1. The first two versions both create a fact that is identified by a single string parameter, with the second version doing so implicitly. The third version also creates a fact identified by a single string, but immediately restricts the domain of the fact to the values given. This means that whereas in the first two versions any string can become an instance of the `citizen` fact, only Alice and Bob can become citizens in the third version. This distinction is known as facts with an infinite or finite domain.

```
Fact citizen Identified by String.
Fact citizen.
Fact citizen Identified by Alice, Bob.
```

Listing 1: Three versions of eFLINT code to declare a new fact.

The fact seen above is known as an `Atomic Fact`, as it consists of one of the two atomic types within the language: `String` and `Int`. The other type of fact is a `Composite Fact`, which is composed out of other types. Unlike the Atomic variant, Composite facts can have more than one parameter. Using this, we can define a fact that stores a citizen and its age, as seen in Listing 2.

```
Fact age Identified by Int.
Fact user Identified by citizen * age.
```

Listing 2: An example of a composite fact within eFLINT.

After declaring a fact, it can be modified by overwriting or extending it. When redeclaring a fact whose name is already present in the program, the new definition of the fact overwrites it. By extending a fact, the original definition is kept, with the new information adding to the definition. Examples of both ways of modifying a fact can be found in Listing 3.

```
Fact age Identified by Int
    Derived from 1, 2, 3.
Extend Fact age
    Derived from 1, 2, 3.
```

Listing 3: Two ways of extending a fact within eFLINT.

Derivation versus postulation. As mentioned before, eFLINT allows for changing the state of the program both through manual actions by the user, and through derivation. Manual changing the state through phrases is known as postulation. Three types of state transitions are possible through these actions: Creating an instance, terminating an instance, and obfuscating an instance. This closely follows the three states a specific instance of a fact can be in: Known to be true, known to be false, or unknown within the knowledge base. The three phrases that correspond to the mentioned transitions are shown in Listing 4.

```
+citizen(Alice).
-citizen(Bob).
~citizen(Charlie).
```

Listing 4: Three phrases that, create, terminate, and obfuscate an instance.

Derivation of instances is achieved by annotating a declaration with derivation rules. These rules indicate how specific instances can be derived, and what conditions must hold in order for the derivation to be possible. The `Derived from` field, as already seen in Listing 3, specifies specific instances of a fact that can be derived. A variant of this field is the `Holds when` field, which is syntactic sugar for the `Derived from` field. It imposes an additional requirement on the derivation, through the use of expressions that must evaluate to true. Lastly, the `Conditioned by` field lists expressions that must always hold in order for a derivation for the fact to be possible. An example can be found in Listing 5.

```
Fact age Identified by Int
    Derived from 1, 2, 3.
Fact old Identified by Int
    Derived from age When age >= 18.
Fact young Identified by age
    Holds when age < 18.
```

Listing 5: eFLINT examples of the possible derivation rules for a declaration.

The derivation rules are used to fill the knowledge base, through a process that is run after every state-changing phrase. This process obfuscates all derivated instances from the knowledge base, and uses an algorithm to compute a new knowledge base using the derivation rules. An important note to consider is that postulation binds more strongly than derivation. Postulated instances are kept intact during the derivation process.

Queries. The knowledge base within an eFLINT program can be exposed through the use of queries. These queries come in two variants: Boolean and instance queries. A Boolean query is meant to evaluate the truth value of a single expression. In the case of instances of facts, this checks if the instance is known in the knowledge base. Instance queries return all the instances that result from evaluating an expression. These instances do not have to hold in the knowledge base to be returned. To filter on only the instances that hold, a variant on the instance query can be used. The three variants can be seen in Listing 6.

```
?citizen(Bob). // Fails or succeeds
?-citizen.     // All citizens
?--citizen.    // All citizens that hold
```

Listing 6: The three query phrases present in eFLINT.

Predicates and invariants. A predicate is a special atomic fact-type declaration that contains only a derivation rule. The predicate can be seen as a query that can be stored for later use, with the expression being the derivation rule. If the state of the program changes, the predicate is reevaluated. An invariant is a special kind of query that must always hold true. If the invariant exists but does not hold true, it is considered violated. An example of both types of predicates can be seen in Listing 7.

```
Predicate bob-is-citizen When citizen(Bob).
Invariant not-old
          When (Forall age : age < 200).
```

Listing 7: The two types of predicates in eFLINT.

Event-type declarations. An event-type declaration is a composite type that can have zero or more parameters. An event can be triggered, which can have effects on the state of the program. These effects are the same as postulation, but can also trigger other declarations. An example event can be found in Listing 8. In this example, `citizen` is the parameter that is given to the event when it is triggered.

```
Event create-citizen
    Related to citizen
    Creates citizen(citizen).

create-citizen(Bob).
```

Listing 8: An example event that creates a citizen instance.

Act-type declarations. An act-type declaration is a variation on an event. It has two extra parameter fields, the `Actor` and `Recipient`. It can be used to describe a relation between the actors in the parameters. Additional parameters can be given through the `Related to` field. Like events, instances for acts can also be derived through derivation rules. An example of an act can be found in Listing 9.

```
Fact candidate Identified by String.
Fact voter Identified by citizen.
Fact vote Identified by citizen * candidate.

Act cast-vote
    Actor citizen
    Recipient candidate
    Holds when voter(citizen)
    Creates vote(citizen, candidate)
```

Listing 9: An example act that creates a vote instance.

Unlike events, acts can be violated. A violation occurs when an act is triggered, without being enabled. An act, like events and duties, can be enabled by creating that instance, either through derivation or postulation. This is identical to the truth value of facts.

Duty-type declarations. Like acts, duties are associated with actors, in this case the duty `Holder` and `Claimant` fields. Unlike events and acts, a duty has no effects and can therefore never be triggered. It can, however, still be violated, through the `Violated when` field present in the duty. When a duty is enabled, it is considered violated if any of the expression in this field hold true. An example duty can be found in Listing 10.

```
Fact election-winner Identified by candidate
    .
Fact administrator.

Duty voting-duty
    Holder citizen
    Claimant administrator
    Violated when election-winner.
```

Listing 10: An example duty that requires users to vote before the end of an election.

Variables and placeholders. Within an expression, variables can occur in the form of identifiers. These identifiers are placeholders for instances of the type they refer to. For example, in the expression `int(x)`, `x` is a placeholder for instances of type `x`. If multiple variables for the same type are needed, they must be denoted through placeholders. A placeholder be explicitly defined through the `Placeholder` phrase. However, implicit placeholders can be formed by adding numbers or single quotation marks after a variable name. An example of both use cases is shown in Listing 11.

```
Fact int Identified by Int.
Placeholder integer for int.

Fact combined Identified by int1 * integer.
```

Listing 11: An eFLINT code snippet that uses both implicit and explicit placeholders.

Constructing instances. To construct an instance of a specific type, all arguments must match the parameters of the type. This means that instances for the `voter` fact can be formed as follows: `voter(citizen(Alice))`. However, in most cases, the inner instance can be simplified, as eFLINT will attempt to typecast the parameters to the correct type. This means that `voter(Alice)` is equivalent to the previous example.

Iterating expressions. Expressions can be split into two types, Boolean and instance expressions. A Boolean expression contains no variables, and as such, evaluating it will result in a single Boolean value. Instance expressions still contain a variable, which has to be filled. Each value for the variable is filled in, resulting in numerous new expressions. This way, the expression is iterated until no variables exist within it.

Alongside implicit iteration, expressions can also explicitly be iterated. This can be done through the use of three iteration operators: `Foreach`, `Forall`, and `Exists`. The `Foreach` operator is the default behaviour when evaluating an expression that contains variables. `Forall` evaluates an expression, and asserts that all instances evaluate to true. The result of this operator is then a Boolean value that can be used in other expressions. `Exists` behaves similarly, but checks if any instance evaluates to true. Three examples are given in Listing 12.

```
+(Foreach age : age).
?(Forall age : age > 18).
?(Exists age1, age2 : age1 + age2 < 20).
```

Listing 12: Examples of explicit iteration within eFLINT.

When iterating over instances, a filter can be applied to the results. This filter can get rid of unwanted instances. To apply a filter to an expression, the `When` keyword can be added as follows: `<EXPR1> When <EXPR2>`. In this example, instances that result from the first expression are filtered out if the second expression does not hold.

4 DERIVATION SEMANTICS

This section expands Section 3 with a focused treatment of a semantics for (fact) derivation in eFLINT. Selecting the semantics has practical consequences on the application of the language and its tooling to use cases. Concretely, it affects (1) which concepts can be expressed, and (2) the performance of implemented tools.

The types of eFLINT be annotated with expressions, defining elements to be derived as a function of which other facts are true or false. Programmers use this to encode logical properties of various forms, for example, property "respect between people is symmetric". As is the case for logic programming languages, these properties are expressed as *(derivation) rules* which encode these properties, but furthermore, afford an operational implementation; given existing facts, a rule can be satisfied by adding new facts. The above example is encoded as rule "if some person 1 respects some person 2, then person 2 respects 1".

Amongst other things, an eFLINT interpreter is tasked with performing a *derivation procedure*, which searches for a model. The contract between interpreter and user can be concretised in their agreement that the model satisfies Properties 1–3, which are stated below. Intuitively, starting from postulated facts, facts are derived if (Property 1) and only if (Property 2) that fact is the result of a rule whose conditions are met, such that all facts have consistent truth-values (Property 3).

Property 1 (Satisfaction). *Each given rule is satisfied, i.e., if each of its conditions are true in the model, then its consequent is true in the model.*

Property 2 (Explainability). *Each true fact was either postulated true, or is derived by a given rule which is satisfied.*

Property 3 (Consistency). *Each model attributes a unique, Boolean truth value to each fact.*

An efficient and straightforward semantics and algorithm is available given a focus on *Horn rules*, which are conditioned only on the *truth* of facts. Our running example was expressed as a Horn rule. What follows (informally) describes what we call the *greedy* algorithm; it works iteratively, starting from an initial set of facts true facts (with all others false by default), those postulated true, and then (1) find a rule whose conditions are met, (2) derive its consequent fact, and (3) repeat until no new derivations are possible. Although simple, the greedy algorithm meets our criteria. Property 3 is preserved by definition. Property 1 follows from looping until no unsatisfied rule remains. Property 2 is preserved as a consequence of a subtle yet vital property: if any rule's conditions are satisfied in some loop, they are satisfied in all future loops, because our assumption prevents the consideration of rules whose conditions become unsatisfied as facts are derived.

In the remainder of the section, we make frequent use of the following three example rule-sets:

1. $p \longleftarrow \neg p$

Figure 2: A ruleset with no stable model. The only rule encodes a contradictory formula of first-order predicate logic.

1. $a \longleftarrow \neg b$
2. $b \longleftarrow \neg a$

Figure 3: This rule set has stable models $\{a\}$ and $\{b\}$; intuitively, the truth of a and b are mutually-exclusive.

1. $a \longleftarrow \neg b$
2. $b \longleftarrow \neg a$
3. $c \longleftarrow \neg c \land b$

Figure 4: This extends the rule set in Figure 3. Rule 3 introduces a logical inconsistency if b is true. Only $\{a\}$ is stable.

Unfortunately, the greedy algorithm violates our properties when applied in a broader context, where rules may be conditioned on the falsity of facts. An illustrative example is given in Figure 2, whose result is model $\{p\}$, violating Property 2; the derivation of p is not explainable by any satisfied rule. Intuitively, p was derived because $\neg p$ was satisfied at some point, but later, this condition became unsatisfied. In this case, the failure of the algorithm is somewhat understandable, as there exists no model that satisfies the rule. Unfortunately, the greedy algorithm is also unreliable in cases where satisfactory models exist. Given the rules in Figure 4, the greedy algorithm may apply rule 2 and then rule 3, terminating with model $\{b, c\}$, violating 2 because b is not explained. The greedy algorithm implicitly makes meaningful decisions in the order it considers applying rules. Here, if it applies rule 2, rule 1 becomes inapplicable, and vice-versa, resulting in different models. Intuitively, the algorithm would need foresight into the consequences of applying rules.

Here, only once rule 3 is applied does it become clear that rule 1 should have been chosen instead of rule 2 in order to ultimately find consistent model $\{a\}$. In general, inconsistencies are detected after arbitrarily many rules and assumptions.

Much literature defines and evaluates semantics reasoning in terms of various notions of negation. The *stable model semantics* [3] is the source of our three properties, calling models satisfying these properties *stable.* Literature on the subject does not guarantee the existence of a stable model for each rule set. In our context, this makes the semantics undesirable, as it would mean that some eFLINT specifications are meaningless, offering no answers to queries. Worse still, specifications appear to be *brittle*; minor adjustments to specifications may make them meaningless. For example, we again consider the rule set in Figure 4. Querying the truth of b yields answer "yes", but this answer disappears with the addition of rule $p \leftarrow \neg p$, despite this rule having no conceptual connection to b. The *well-founded semantics* [19] takes an alternative approach that ensures that each rule set has exactly one model, but at the cost of removing Property 3. Effectively, there is still not always a Boolean answer to a given question. However, the well-founded semantics is designed such that its facts' truth values coincide with those of the unique stable model, if it exists. This semantics is detailed in [19]. Here, it suffices to say that we draw inspiration from its guarantee that every rule set has a model at the cost of it preserving only weakened versions of Properties 1–3.

Our semantics for eFLINT is operationalized by an adaptation of the greedy algorithm. Intuitively, it preserves Property 2 at the cost of preserving Property 1; rules which "directly lead to logical inconsistencies" are discarded, and only the remaining rules are satisfied. The new algorithm, which we will call the *tracking* algorithm, explores the search tree of rule-applications. As before, the model is built incrementally, by applying one rule at a time. Unlike before, when rules are applied, their negative conditions are stored as *assumptions.* When a derived fact conflicts with an assumption, the search *backtracks* to the state before the rule was applied, and considers another rule instead. Backtracking rules are not (immediately) considered again, and so, they are effectively discarded.

The result is a weakening of Property 1 as the model may satisfy the conditions of a rule despite not deriving its consequence (as doing so results in inconsistencies!). For example, consider a possible execution when applied to the rule set in Figure 4; at each step, we call the-derived true facts T and the assumed-false facts F:

(1) Initially: $T = \{\}, F = \{\}$,
(2) apply rule 2: $T = \{b\}, F = \{a\}$,
(3) apply rule 3: $T = \{a, b\}, F = \{a\}$,
(4) backtrack to before step 2 and apply rule 1, and
(5) terminate with: $T = \{a\}, F = \{b\}$.

The model returned by the tracking algorithm is encoded in T: facts in T are true, and all others are false. As with the greedy algorithm, Property 3 is preserved by definition. Unlike the greedy algorithm, the effects of each applied rule persist throughout the search; its conclusion and positive conditions are retained in T as before, but now, the negative conditions are retained in F, such that the model satisfies all conditions of applied rules. Ergo, Property 2 is preserved. This is meaningful to users; intuitively, every true fact is explained by an applicable rule, and in turn, the applicability of each rule is explained by the satisfaction of its conditions. To make this possible, the tracking algorithm discards rules in contexts found to introduce logical inconsistencies. The nature of the algorithm makes it possible to draw these discarded rules to the attention of the user, perhaps with the derivation sequence that resulted in an inconsistency. Strictly speaking, this means that Property 1 is not preserved; users cannot rely on the model satisfying a given rule. In practice, the discarded rules occur only as a result of circular definitions, which are not expected to arise intentionally. Users can then use the results to repair the logical inconsistency. For example, the following eFLINT snippet appears to encode a sensible default assignment of Amy as leader. However, it encodes an unintuitive logical inconsistency: Amy's leadership is predicated on the absence of Any leaders (including herself!).

```
Fact leader Derived from leader(Amy)
  When !(Exists leader: leader).
```

This notion can be modelled in a logically consistent manner by the following, adjusted eFLINT snippet:

```
Fact leader Derived from leader(Amy)
  When !(Exists leader: leader
                     && leader != Amy).
```

Evaluation. This final version of this paper is intended to compare and contrast the discussed alternatives as candidates for giving semantics to eFLINT derivation rules. The pros and cons will be discussed in relation to requirements formulate for the language, obtained through the various application areas of eFLINT.

5 REASONER ARCHITECTURE

This section proposes a client-server architecture for normative reasoning in which the server interprets eFLINT programs. The architecture is intended to be used as a generic blueprint for specialised architectures for specific applications in which clients can vary from user interfaces to back-end components for automated access control, runtime verification and auditing services. Central to the architecture is the reasoning API (given as a JSON specification) that standardises communication with the reasoner server. In this version of the paper we focus on a specific version of the architecture with a user interface for executing eFLINT phrases. In the full version of the paper we will also include one or more architectures that demonstrate software actors interacting with the reasoner.

The architecture presented in Figure 5 is for a system designed as a set of components running in parallel and communicating via message-passing. The architecture decouples logical components such that their implementations and host machines may be specialised to their tasks. There are three components at the root. (1) Each *client* interacts with a user, parses their input in the eFLINT language, sends language fragments as updates to the server and renders server output for the user. (2) The *server* interacts with clients, digesting language fragments, performing derivation, and answers queries. It delegates the storage of rules and facts to the database. (3) The *database* stores persistent rules and facts. The database is optional as the server can be used in a stateful mode in which

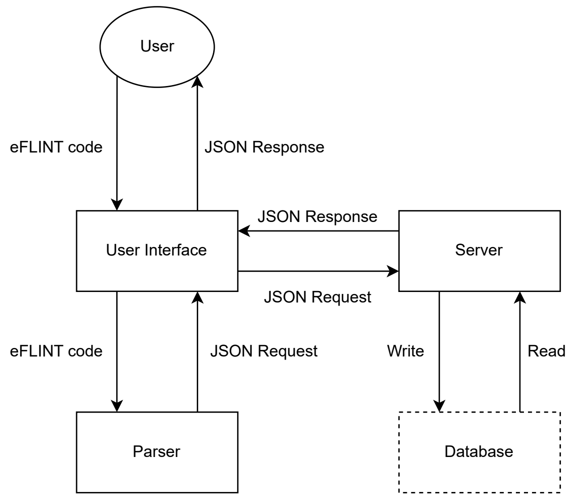

Figure 5: A topology of the interaction between various components in the resulting application. The database is dotted out, as the server currently stores the state within the program. The left-hand side of the topology is located client-side, the right-hand side is located server-side.

the current knowledge base (and information regarding types) is maintained in RAM. Alternatively, the server can use a database to maintain program state in a way that ensures persistence, making the application resilient to server crashes. Furthermore, the API is designed such that server responses contain a list of phrases that together summarise the effects of the incoming request. The phrases can be used by the client to keep its own record of effects and reproducing program state.

The server itself is a system of communicating components as is described in more detail in Section 6.

5.1 JSON Specification

To standardise communication with the server, an API has been designed and formulated as a JSON specification. The specification is available online [9].

The specification describes both directions of the communication between a client and the server. Firstly, the mapping from eFLINT to the JSON format is documented. This is done by introducing JSON schemas that correspond to the phrases of eFLINT. Expressions are often recursive, taking the form of an Abstract Syntax Tree (AST) that can be used to describe all expressions in the eFLINT language. As an example, the following eFLINT phrase could be used to create an instance for the fact `int`: `+int(6 - 2)`. Translating this into the JSON specification results in the JSON code seen in Figure 6.

Secondly, the specification specifies the response format from the server. Every phrase sent to the server is responded to with a `phrase-result` type. This type indicates if the phrase was executed successfully, and what effect the phrase had in terms of state

```
{
    "kind": "create",
    "operand": {
        "identifier": "int",
        "operands": [
            {
                "operator": "SUB",
                "operands": [6, 2]
            }
        ]
    }
}
```

Figure 6: The JSON specification for the eFLINT phrase `+int(6 - 2)`.

changes and violations. As an example, an interaction with the server is shown in Figure ??. This figure shows an eFLINT code snippet and the corresponding JSON request and response from the server.

6 INTERPRETER IMPLEMENTATION

This section details the implementation of the critical components of the new eFLINT interpreter, introduced in Section 5. First, we detail the (de)serialization of eFLINT phrases for representations, specialised to the communication between user and client, and between client and server. Second, we detail the new derivation procedure, implementing the new semantics explained in Section 4. Finally, we detail two key design choices made in implementing the new interpreter, that aim to improve execution time or memory usage.

6.1 Communication between the components

In Figure 5 the flow of communication between the user, client and server can be found. An important distinction to make between these forms of communication, is that the user-client communication happens locally, while the client-server communication does not. The user can send plain eFLINT code to the client, which then serializes the code to a JSON format compliant to the JSON specification using a parser. This JSON can then be sent to the server, which in turn deserializes it into an internal representation.

6.2 New Derivation Procedure

In order to test with different types of derivation semantics, the derivation procedure in the new interpreter is largely modular. The procedure can be swapped out for a different one, to see the effect on the behaviour of the reasoner. To implement the new derivation semantics, as discussed in Section 4, a new module can then replace the original derivation procedure.

The implementation follows a backtracking approach, using a map as a way to store all assumptions made during the derivation process. This map links assumptions to the state of the derivation process at the point the assumption is made. Then, whenever a new instance of a fact is derived, the map of assumptions is checked. The presence of the new instance as a key in the map implies that this instance was once assumed to be false, and is now derived to be true. If this occurs, the procedure replaces the current state of the program with the state stored in the map, thus backtracking to the assumption.

6.3 Interpreter Optimisations

An increase in *performance* was one of the goals for the new interpreter, when comparing it to the existing Haskell interpreter. To measure this performance, execution time and memory usage were chosen as two key factors. To achieve this performance gain, certain design choices have been taken. This subsection aims to draw attention to two of these choices.

6.3.1 Hashing. To increase both execution time and memory usage, hashing has been used to store instances of facts. Instead of always storing the entire instance, a hash of the instance is stored whenever possible. This way, the original instance can no longer be retrieved, but still be compared with other instances.

When the entire instance of a fact needs to be stored, as it may need to be accessed at a later point, hashing can still be used to improve the execution time of the interpreter. This is done by storing the instances in a map that acts as a set, with the hash as the key and the instance it represents as the value. This allows for amortized *O(1)* lookup, addition and removal, while still being able to enumerate all known instances.

6.3.2 Concurrency. As the new interpreter is written in Golang, a language supporting concurrency through goroutines and channels, an attempt was made to use concurrency within the new interpreter. The most useful area where concurrency seemed to show an improvement in performance, was in the evaluation of expressions. Within eFLINT, the evaluation of an expression can lead to several new expressions. When handling a phrase in the interpreter, the state of the program can only be modified after every expression within the phrase is evaluated. Otherwise, the evaluation of later expressions can be influenced by a change in state through a previous expression, while both expressions are part of the same phrase. This means that while concurrency can be used, all end results need to be stored, such that the state is modified only at the very end. This limits the usability of concurrency in this area.

However, this limitation is not present when evaluating expressions that are part of a derivation procedure. This is because the derivation procedure repeats itself until stable, so changing the state mid-way evaluation leads to the same result as changing the state after each step of the derivation procedure. This realisation allows for expressions to be dynamically evaluated, through the use of goroutines. Dynamic evaluation is faster than static sequential evaluation, as all expressions could theoretically be evaluated at the same time. Moreover, while this form of concurrency brings overhead, the total memory usage will go down as there is no need to store all evaluated expressions in memory. As soon as an expression is evaluated and used, it can be discarded, freeing memory. The final version of the paper will discuss additional implementation strategies.

6.4 Evaluation

To properly test the created eFLINT interpreter, it needs to be checked on both correctness and performance. Correctness tests must cover the entire eFLINT language, to ensure that the new interpreter behaves as intended. Performance-wise, specific parts of the eFLINT language must be tested at various levels of computational cost. By comparing the performance with the original interpreter, specific areas within the design can be found that perform better or worse in the new implementation.

This subsection focuses on the setup of these experiments, what tests will be run, and what specific measurements will indicate success / failure. The results of the tests are also presented.

6.4.1 Experimental setup. As Golang has been used to implement the eFLINT server, the built-in Golang unit and benchmark testing tools have been used. These tests allow testing for both correctness and performance. The Haskell reference interpreter has also been tested using the Golang testing framework, through the use of the `eFLINT REPL`.

The Golang benchmarking testing framework runs a test multiple times, creating reliable measurements. The framework reports the average time in nanoseconds the function took over the number of times it has been tested. However, it does not report the standard deviation of the test. Due to this limitation, error bars have not been included in the figures.

The framework allows for setting a timeout, after which any test will fail due to running too long. When benchmarking the performance, this limit has been set to one hour. Specific values of tests that are cut off this way are omitted from the results.

The reported experiments use the following hardware and software. The machine has an Intel Core i5-9300H with 4 cores, running at a 2.4GHz base clock frequency, and has 8Gb of RAM available. The tests were run on Ubuntu 22.04, with Golang 1.18.1.

6.4.2 Correctness evaluation. To test the correctness of the new implementation, 42 test programs have been put together [2]. Together, these programs are intended to cover the full eFLINT language, each testing a specific aspect or construct of the language. The tests employ Boolean queries as assertions to verify that a program is interpreted correctly. The testing framework assumes that all queries are written such that they hold (yield true) upon success.

Although practical, the Boolean queries of eFLINT can only be used to verify properties related to the single, 'current' knowledge base at that moment in the execution of a scenario. Although not reported in this work, experiments have been performed with LTL formulae that can be used to describe more abstract properties and express properties over sets of traces rather that individual knowledge bases.

6.4.3 Performance evaluation.

Derivation rules. The first test performed is scaling the amount of derivation rules that must be evaluated to fill the knowledge base. To this end, the eFLINT pseudocode in Listing 13 has been used. The size of the domain of x increases exponentially in each iteration of the test, starting at 1 and ending at 10000. Due to the

[2] https://github.com/Olaf-Erkemeij/eflint-server/tree/main/cmd/eflint-server/tests/correctness

way the code is set up, each iteration of the derivation rules can only add a single value of x to the knowledge base.

```
Fact x Identified by 1..<size>
  Holds when x(x - 1).
+x(1).
```

Listing 13: The template code used for running the derivation rule test. The value for <size> is filled in with the desired domain size.

The result of this test can be seen in Figure 8 and Figure 9. The reference interpreter starts out with near constant time, but then quickly jumps in execution time to not being able to complete the last test. The new interpreter shows that runtime and memory consumption seem to scale linearly with the tested size of the domain. Concerning memory usage, the reference interpreter stays linear at first, but then increases slightly as the domain size increases. Our work again shows linear scaling, starting out at a higher initial value than the reference interpreter.

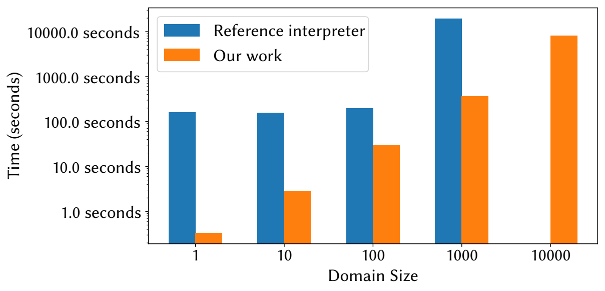

Figure 8: The average execution time for running the derivation rules test for both eFLINT interpreters. The last measurement is omitted for the reference interpreter, as the program did not finish in time.

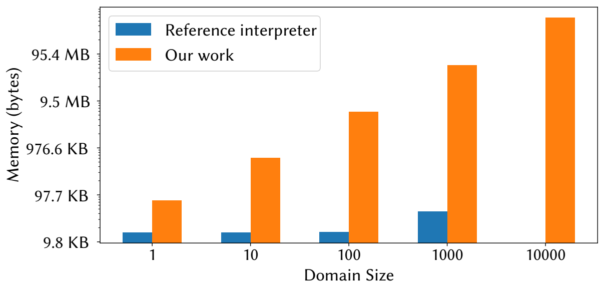

Figure 9: The average memory usage for running the derivation rules test for both eFLINT interpreters. The last measurement is omitted for the reference interpreter, as the program did not finish in time.

The final version of this paper is intended to compare alternative implementations of derivation semantics for performance.

Composite type dimensionality size. The second test performed is on scaling the dimensionality of a composite type, and then listing all the possible instances of this type. To test this, the eFLINT pseudocode in Listing 14 was used. The number of parameters starts at 1, and is increased until a total of 5 parameters are used for the composite type.

```
Fact param Identified by 1..10
Fact combined
  Identified by param1 * param2 * ...
?-combined.
```

Listing 14: The base code used for running the dimensionality size test.

The result of this test can be seen in Figure 10 and Figure 11. While the reference interpreter seems to show a constant execution time, with some growth in the later iterations of the test, our work grows linearly with the parameter size. As for memory, both interpreters show linear growth in usage, but our work starts off with a higher amount of memory used.

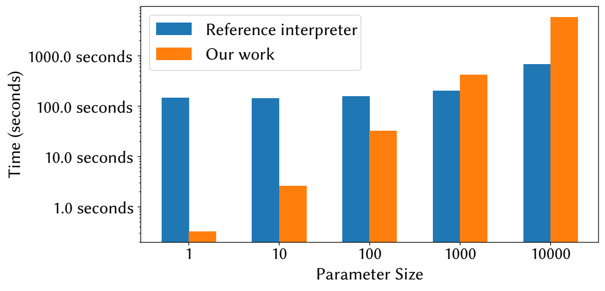

Figure 10: The average execution time for running the composite type dimensionality size test for both eFLINT interpreters.

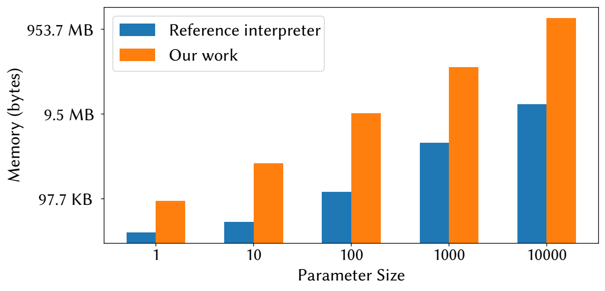

Figure 11: The average memory usage for running the composite type dimensionality size test for both eFLINT interpreters.

7 CONCLUSION

The eFLINT language has been successfully applied in various research projects to demonstrate the benefits of automating compliance using a domain-specific language and a dedicated normative reasoner. To bring the language to practice, however, standardisation is required of both the syntax and semantics of the language as well as the communication with the normative reasoner. This paper gives a detailed description of the latest version of the language, supported by a reference interpreter.

The paper has further discussed the implementation of a novel interpreter, marking a step towards the more efficient and scalable execution of eFLINT programs. The interpreter forms the basis of a normative reasoner with a standardised API developed to support various scenario's for automated compliance. The full version of the paper will discuss additional optimisation strategies and present additional empirical analyses. The full version of the paper is also intended to evaluate the alternative derivation semantics presented in the paper against laid out requirements and for performance.

ACKNOWLEDGMENTS

This work has been executed as part of the SSPDDP project supported by NWO in the Big Data: Real Time ICT for Logistics program (628.009.014) and the AMdEX Fieldlab project supported by Kansen Voor West EFRO (KVW00309) and the province of Noord-Holland.

REFERENCES

[1] Keith L. Clark. 1978. *Negation as Failure.* Springer US, Boston, MA, 293–322. https://doi.org/10.1007/978-1-4684-3384-5_11

[2] Christopher A. Esterhuyse, Tim Müller, L. Thomas van Binsbergen, and Adam S. Z. Belloum. 2022. Exploring the Enforcement of Private, Dynamic Policies on Medical Workflow Execution. In *18th IEEE International Conference on e-Science, e-Science 2022, Salt Lake City, UT, USA, October 11-14, 2022.* IEEE, 481–486. https://doi.org/10.1109/eScience55777.2022.00086

[3] Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics For Logic Programming. *Logic Programming* 2 (12 1988).

[4] Wesley Newcomb Hohfeld. 1917. Fundamental Legal Conceptions as Applied in Judicial Reasoning. *The Yale Law Journal* 26, 8 (1917), 710–770. http://www.jstor.org/stable/786270

[5] Alfred Horn. 1951. On Sentences Which are True of Direct Unions of Algebras. *J. Symb. Log.* 16, 1 (1951), 14–21. https://doi.org/10.2307/2268661

[6] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76. https://doi.org/10.1145/3338843

[7] J. Jaffar and J.-L. Lassez. 1987. Constraint Logic Programming *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 111–119. https://doi.org/10.1145/41625.41635

[8] John W Lloyd. 2012. *Foundations of logic programming.* Springer Science & Business Media.

[9] Tim Müller, L. Thomas van Binsbergen, and Olaf Erkemeij. 2022. *eFLINT JSON Specification.* https://gitlab.com/eflint/json-specification

[10] Mostafa Mohajeri Parizi, L. Thomas van Binsbergen, Giovanni Sileno, and Tom M. van Engers. 2022. A Modular Architecture for Integrating Normative Advisors in MAS. In *Multi-Agent Systems - 19th European Conference, EUMAS 2022, Düsseldorf, Germany, September 14-16, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13442)*, Dorothea Baumeister and Jörg Rothe (Eds.). Springer, 312–329. https://doi.org/10.1007/978-3-031-20614-6_18

[11] Teodor Przymusinski. 1990. Well-Founded Semantics Coincides with Three-Valued Stable Semantics1. *Fundamenta Informaticae* 13, 4 (Oct. 1990), 445–463. https://doi.org/10.3233/FI-1990-13404

[12] 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).* 364–369. https://doi.org/10.1109/RE48521.2020.00049

[13] L. Thomas van Binsbergen. 2023. *eFLINT Jupyter Notebooks.* https://gitlab.com/eflint/jupyter

[14] L. Thomas van Binsbergen. 2023. *Reference implementation of eFLINT in Haskell.* https://gitlab.com/eflint/haskell-implementation

[15] L. Thomas van Binsbergen, Milen G. Kebede, Joshua Baugh, Tom van Engers, and Dannis G. van Vuurden. 2022. Dynamic generation of access control policies from social policies. *Procedia Computer Science* 198 (2022), 140–147. https://doi.org/10.1016/j.procs.2021.12.221 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks / 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare.

[16] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. 2020. eFLINT: A Domain-Specific Language for Executable Norm Specifications. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Virtual, USA) *(GPCE 2020).* Association for Computing Machinery, New York, NY, USA, 124–136. https://doi.org/10.1145/3425898.3426958

[17] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2020. A Principled Approach to REPL Interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020).* Association for Computing Machinery, New York, NY, USA, 84–100. https://doi.org/10.1145/3426428.3426917

[18] Maarten H. van Emden and Robert A. Kowalski. 1976. The Semantics of Predicate Logic as a Programming Language. *J. ACM* 23, 4 (1976), 733–742. https://doi.org/10.1145/321978.321991

[19] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. 1991. The well-founded semantics for general logic programs. *J. ACM* 38, 3 (July 1991), 619–649. https://doi.org/10.1145/116825.116838

[20] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96. https://doi.org/10.1017/S1471068411000494

Received 7 August 2023