Traits: An Object Oriented Dynamic Type System for Reasoning with Unstructured Data in a Type Safe Environment.

Davide Sottara[1][0000−0002−6746−4710], Mark Proctor[4,5][0000−0002−1746−072X], Stefano Bragaglia[3][0000−0002−2609−0169], and Mohammad Hekmatnejad[2][0000−0003−0110−0006]

[1] Biomedical Informatics Department, Arizona State University, 13212 East Shea Boulevard, Scottsdale, AZ 85259, USA
[2] Computer Science Department, Arizona State University, Tempe, AZ 85281, USA
[3] Context Scout, London (UK)
[4] Dept. of Electrical & Electronic Engineering Imperial College London, London (UK)
[5] Red Hat, Westford, MA (USA)§

Abstract. This paper describes "traits", an extension to a rule engine that provides a dynamic type system for working with in an object oriented code in a type safe environment. This extension is inspired by frame logic and "duck typing". The implementation is based on the use of dynamic proxies and form of runtime interface injection and requires the partial redefinition of the traditional working memory operations (assert, update, retract), but is otherwise transparently embedded in the engine. We evaluate a reference implementation built on top of the open source rule engine Drools, showing that the approach improves clarity without impacting performance.

Keywords: Production Rule Systems · Rule Engine · Frame Logic · Drools
.

1 Introduction

The claim of this paper is provide production rule systems with a dynamic type system that is also type safe. The type system must be compatible with an object-oriented context with inheritance and polymorphism, where it is possible to distinguish between classes, types and fields (also called attributes, properties or slots). Dynamically typed systems work more effectively with loosely unstructured data, but lose the benefits of type safety. A statically typed object orientation, allows for more robust systems [25], but does not cope well with a dynamic domain, since types are assigned when an entity is created and can not be reevaluated. A solution is proposed that combines type safety with dynamic typing even in the context of mainstream OO languages that do not support it. A reference implementation based on the rule engine Drools [21] is provided; the benefits and limitations of the approach are further discussed

in Section 5, based on some theoretical considerations and the results of some empirical benchmarks.

The underlying principles are founded on an adaptation of Frame Logic (F-Logic) [14] to production rule systems [12], leveraging the automatic management of dynamic, transparent proxies [24] that facilitate interface injection. The classic RETE engine behavior has been extended so that rules can be written against objects or proxies alike in a transparent manner. Any modification of an object, due to the execution of a rule's consequence, is guaranteed to be reflected by its current proxies supporting its virtual types and vice versa. The implementation supports both hard and soft properties in a uniform and transparent way. Hard properties are provided by the underlying class system, such as fields in Java, and provide for efficient and compact memory storage but cannot be dynamically changed. Soft properties are provided by the framework, and supplement dynamic properties at the instance level.

In the rest of the paper, the notation used to formalize the behavior is also borrowed from F-Logic. Classes and individuals are denoted using constant symbols. The symbol : denotes a membership assertion (e.g. joe : `Person`), while :: denotes class subsumption (e.g. $DiabeticPx :: Patient$). Fields are modelled as (inheritable) class properties, and their name and type(s) are linked using the symbol $\Rightarrow$ within the scope of a class, such as in $Patient[mrn \Rightarrow \{String, Identifier\}]$. Finally, the symbol $\rightarrow$ is used to assert field values.

2 Related Works

The challenges of combining loosley structured data with a dynamic type system that is also type safe are well known and have been explored in a multitude of systems. The concept that historically is more similar to what is proposed is that of "traits". This term is adopted for the dynamic type system, even if with some differences with respect to other systems. The term was first mentioned in [17], where it is referred to as the "Traits Mechanism". The paper was based on an implementation for the Star Graphics system and the mechanism was based on the idea of Simula-67 classes: where Simula-67 only allowed single inheritance, traits in Star Graphics allowed for multiple inheritance. It defined traits as "a characterization of behaviour and ... the primitive abstraction used to define objects". Star Graphics traits included, among other things, both state and operations, which exposes the "diamond problem" of multiple inheritence when multiple traits provide different implementations for the same operation name. The Star Graphic's class Trait was responsible for creating and destroying objects in that class, acting as a factory. Once instantiated, an object's trait hierarchy was fixed and slots for data must be a part of the defined trait.

The term traits has since been used in a number of software systems, where its definition has changed and evolved. Smalltalk created the concept of "roles" as a variation on traits [10], but remains a statically typed system. In [23] traits are further refined to only include operations, aiming to produce a system for fine grained, composable units of behaviour. The Self programming language

[26] adopts a prototype approach: slots can be added over time and the type can be dynamically changed. In this proposed system, objects can be instantiated without the need of a trait class as a constructor, and properties (state) can be added at runtime without any additional association. This allows for unstructured data at runtime, as differentiated from the static structured data of the Star Graphics system. Traits only involve representing state, realised in the terms of properties, which allows to manage multiple inheritance issues as will be described later in this paper. At runtime, an object plus a conforming set of properties can thus be associated with one or more traits, through a process of type recognition, providing class-like object oriented access to otherwise unstructured data.

The static binding used e.g. in Java can be considered a trivial type of recognition: all objects of class `C` are automatically recognized as having type T when `C` implements T. Signature compliance or "duck typing" [8] is another basic recognition method: in order for an object to have type T, it is necessary and sufficient that its class (public) signature be a superset of the interface's signature. Duck typing does not require to enforce sound semantics, so more advanced and robust strategies should be adopted. (Business) rules are commonly used for decision making, based on the identification of complex situations, involving a number of instances matching specific patterns. Individual classification is also one of the main problems of Description Logic [4], where types are recognized matching an object's description with a concept's definition. Even if most of the methods are compatible to some degree with a rule engine implementation, the actual discussion of the best recognition strategy is beyond the scope of this paper. Instead, the paper focuses on the problem of using objects with dynamic types for decision making, once the classification criteria have been applied.

3 Proposed approach

Conceptually, the framework makes a distinction between rigid type classes (e.g. `Person`) and role type classes (e.g. *Patient*), improving support for the latter when roles are either recognized, acquired or relinquished at runtime, rather than being asserted statically. In order to work with objects directly, rather than with their representation, it is necessary to support an operation called interface injection [22]. Unlike the Java specification, however, it is required that individual instances can be injected independently: i.e. if a particular `Person` is recognized to be a *Patient* and should be treated as such, the same should not hold true for all other `Person`s in general. Allowing individual type injection raises two major issues: an object's class may not provide an implementation for all the methods declared in the injecting interface and multiple injections may lead to multiple inheritance conflicts [7]. Traits, in general, may overlap, while the underlying object model is statically typed. The seemingly obvious option of making *Patient* a subclass of `Person` is disregarded. In fact, if Θ is the set of types, the engine would need a class for each element of the power-set $\mathcal{P}(\Theta)$. Moreover, an object would have to be destroyed and reinstatiated every time its

current set of types θ changes. Alternative options common in the database world such as variant classes with type flags are also impractical, since they would unnecessarily expose attributes, most of which would be irrelevant, and possibly lead to inconsistencies (e.g. setting the diabetes type for an `OncologyPatient` who is not actually suffering from diabetes). Using separate helper objects, one for each actual type, and linking them to the original object is a more scalable approach, since up to $|\Theta|$ additional objects would be required for each individual. However, this solution does not preserve transparency (e.g. both a `Person`'s `OncologyPatient` and a `DiabeticPatient` proxy would qualify when looking for a *Patient*). With interface injection, instead, the type set Θ could be defined as a taxonomy of interfaces which can be applied (and removed) dynamically to individual instances of a class, part of an information model Γ, preserving types and identities at the same time. Interface injection is not supported natively by the underlying object system. This is emulated by a combination of interfaces and dynamic, wrapper proxies similar in principle to what has been done in [24], the management of the proxies is delegated to the inference engine, making it transparent to the user.

3.1 Traiting semantics

Type and Class definitions. In JavaClasses and objects are separate entities, so there is no support for reification of classes as individuals, nor non-inheritable class properties (i.e., static properties). This is an important restriction over the model defined in [14]. An immediate consequence is that class and type hierarchies are statically asserted and no new subsumption relationship or property membership can be inferred at runtime: if necessary, this step should be executed during a pre-processing phase. A distinction is made between a conceptual domain model Θ and an information model Γ, typically used to deliver data representing the domain, and it is assumed that the former will be implemented using interfaces, while classes will be used for the latter. A type (interface) $T \in \Theta$ is then defined by its name T and zero or more ancestry assertions $T :: S_j$ for some $j \geq 0, S_j \in \Theta$. Moreover, a type will have zero or more inherited, typed properties $p_k(T), k \geq 0$. A property p is defined by its identifier, usually p itself, and its range type $X \in \Theta \cup \Gamma \cup \Delta$, where Δ is a set of primitive data types (strings, integers, dates, etc ...). The range can be scalar or collection oriented. Some properties may also have an inheritable default value, compatible with the property type defined in the property signature, which is used in case no other value is provided. It is further assumed that signatures will be defined for all properties. So, for example, it could be defined $Patient[\ name \Rightarrow \texttt{String};\ address \Rightarrow^{\star} \texttt{Address};\ contactPerson \Rightarrow \texttt{Person}\ ]$ and $DiabeticPatient[\ diabetesType \rightarrow II\ ] :: Patient$. Classes from the information model Γ are defined similarly, except that each class can have at most one parent. Classes and interfaces may have additional methods, but the focus is only on their properties' signatures and inheritable (default) values.

Except for inheritable default values of class properties on type interfaces, this representation can be implemented directly in Java and similar languages,

provided that scalar properties $p \Rightarrow X$ are interpreted as the accessor pair $T[\ getP \Rightarrow X;\ setP\ @\ X \Rightarrow \bot\ ]$ (resp. for collection-oriented ones). There is explicit need to preserve the immutability of classes and types to retain the type safety benefits of code compilation but, unlike Java, it is not assumed that types are statically determined by the class which an object belongs to.

Object-Type association Given an instance $\circ$ of a class $\mathtt{C} \in \Gamma$ and a type interface $T \in \Theta$, interface injection is semantically equivalent to the direct, runtime assertion $\mathtt{o} : T$, as opposed to inferring it statically from $\mathtt{o} : \mathtt{C} \wedge \mathtt{C} :: T$. Such assertions can also be retracted, on an individual basis. Unfortunately, multiple such assertions can introduce multiple inheritance and possibly lead to an inconsistent state when an object is instance of at least two disjoint types, or (recursively) has a property slot filled by a value that is inconsistent. Formally, given $\mathtt{o} : \mathtt{C} \wedge \mathtt{o} : T_j$ (asserted or inferred), no two types T_1 and T_2 must be disjoint. As in frame logic [14] or description logic [4], disjointness is not assumed by default. Asserting disjointness axioms is currently not supported, but inconsistencies can arise with properties. Given a property $p \Rightarrow X$ defined by T, with optional return value $p \rightarrow x$, inconsistencies can arise if:

1. p may not be defined or inherited by $\mathtt{C}$
2. p may be defined or inherited by $\mathtt{C}$ with a different signature $p \Rightarrow Y$
   (a) p may be read through $\circ$ or the interface associated to T, invoking either $\mathtt{C}[getP]$ or $T[getP]$.
   (b) p may be updated using either $\mathtt{C}[setP]$ or $T[setP]$.
3. p may be defined or inherited by $\circ$ with the same type but $p \rightarrow y$ and $x \neq y$

The first conflict can be solved allowing the property set of an individual object to be extended in a controlled way. Avoiding the other two requires to impose some additional constraints on the injection of a type into an existing object. The details are discussed in the next paragraph.

Class-Interface Mapping An object $\circ$ has a set of preallocated slots $\mathbf{p}_\mathtt{C}$ derived from its concrete class $\mathtt{C}$ and its ancestors. Whenever a type is asserted, the interface is required to provide a set of slots $\mathbf{p}_T$. In statically typed systems such as Java, $\mathbf{p}_T \subseteq \mathbf{p}_C$ is enforced at compile time. This proposal does not make such assumption: the set $\mathbf{p}_\mathtt{o}$ of properties available to an object is, at any time, the union of all the properties inherited through the currently asserted types. So, $\mathbf{p}_\mathtt{o} = \mathbf{p}_\mathtt{C} \cup_{j|\mathtt{o}:T_j} (\mathbf{p}_{\mathtt{T}_j})$. Moreover, a property $p \in \mathbf{p}_\mathtt{o}$ is called a *hard field* iff $p \in \mathbf{p}_\mathtt{C}$, a *soft field* otherwise. The former are directly part of the object's implementation, while the latter must be allocated by the framework.

Following the approach in [14], for each property $p \in \mathbf{p}_\mathtt{o}$ its range type is taken to be the intersection of all the range types defined by the object's class and its current interfaces:

$$o[\ p \Rightarrow \bigwedge_i Y_i\ (\in \Gamma \cup \Theta \cup \Delta)\ \mid o : Z_i\ (\in \Gamma \cup \Theta) \wedge Z_i[\ p \Rightarrow Y_i\ ]\ ] \qquad (1)$$

In the case of soft fields, no range type definition exists in the concrete class, so it is taken it to be $\top$ (i.e. only those interfaces that define the soft field are considered). Given a candidate object $\mathtt{x}$ to fill the slot p of $\circ$, this requirement can be interepreted either as an axiom or as a restriction. More formally, in the former case $\exists i : (\mathtt{o} : Z_i \wedge \neg(\mathtt{x} : Y_i)) \Rightarrow \neg p(\mathtt{o}, \mathtt{x})$, while in the latter $\forall i : (\mathtt{o} : Z_i \wedge p(\mathtt{o}, \mathtt{x})) \Rightarrow \mathtt{x} : Y_i$. A mixed approach is adopted: whenever the range type $Y_i \in \Gamma \cup \Delta$ is a class or a datatype, it is assumed to be a constraint:

$$\mathtt{o} : \mathtt{Z} \in (\Theta \cup \Gamma) \wedge \mathtt{Z}[p \Rightarrow X \in (\Gamma \cup \Delta)] \wedge \neg(\mathtt{x} : X) \Rightarrow \neg p(\mathtt{o}, \mathtt{x}) \tag{2}$$

That is: since only interface types can be assigned dynamically, whenever a property requires to be filled by a class (or datatype) X, no individual $\circ$ can be assigned unless it already belongs to that class, otherwise the knowledge base would become inconsistent. When the range is an interface, instead, the ability to make runtime assertions allows to support either modality in a mutually exclusive way:

$$\mathtt{o} : \mathtt{Z} \in (\Theta \cup \Gamma) \wedge \mathtt{Z}[p \Rightarrow X \in (\Theta)] \wedge \neg(\mathtt{x} : X) \Rightarrow \neg p(\mathtt{o}, \mathtt{x}) \tag{3a}$$

$$\mathtt{o} : \mathtt{Z} \in (\Theta \cup \Gamma) \wedge \mathtt{Z}[p \Rightarrow X \in (\Theta)] \wedge p(\mathtt{o}, \mathtt{x}) \Rightarrow \mathtt{x} : Y \tag{3b}$$

Notice that whenever a new type is injected or removed from $\circ$, the objects filling its slots should be constrained or updated accordingly, but the operation can be executed incrementally since a conjuction of assertions or constraints is associative.

Default property value initialization An object's slots are usually initialized during its construction, or assigned explicit values later using accessors. If no value for a property is explicitly given or set, default values may be inherited from ancestor classes or (dynamic) interfaces. A sound strategy to deal with inheritable class properties $\mathtt{Z}[\, p \rightarrow v \,]$ is presented in [27]. 3-valued logic is not introduced into the framework (to be integrated and investigated in a future work), but nevertheless adopt a similar cautious approach. The most specific inheritance contexts for p are considered, i.e. the set $\{\mathtt{Z}_j \mid \mathtt{Z}_j[\, p \rightarrow v_j \,]\}$, such that $\mathtt{o} : \mathtt{Z}_j$ and there is no $\mathtt{Z}_k | \mathtt{o} : \mathtt{Z}_k \wedge \mathtt{Z}_k :: \mathtt{Z}_j$. In general, a number of candidate values $\{v_j\}$ will be retrieved: if there is only one candidate v_0, that value is chosen to populate the slot, i.e. $p(\circ, v_0)$ is asserted. Otherwise, the conflict is resolved by leaving the value unspecified.

Property value access To guarantee type safety, methods have to be partially anti-monotone [14] with respect to their input and output arguments. Given a property p, this requirement holds for its accessors, getters $\mathtt{o}[\, getP \Rightarrow Y \,]$ and setters $\mathtt{o}[\, setP \mathbin{@} Y \Rightarrow \bot \,]$, for all class and types Z_i that (re)define p with some type Y. By definition (1), given a property p, any valid object in its range must have all the required types, so it can be accessed safely using any of the getters. Setters are not so straightforward, since when a particular setter is used, the new value will be guaranteed to be compatible with at least one, but not all, of

the required types. So, any valid implementation of a setter will have to apply either (2) or either one of (3a) and (3b) before any slot can be succesfully updated. Setting the value of a field is considered an explicit (re)assertion at the object level [27], so it will override any previous or inherited default value. However, should the newly set value violate either (2) or (3a), the property p will be reset to its current default value (if one exists), or left unspecified.

3.2 Architectural Overview

While the model described in section 3.1 originated in a logic programming context, it has been specialized to be compatible with an object-oriented framework in general and a Java-based environment in particular. Any type in Θ is mapped to a Java interface with the same name, which exposes the required properties as pairs of accessors. All interfaces are annotated with a Java marker annotation and extend a known interface *Top*, which models $\top$, as well as the internal interface *CodedType*, which provides the internal data structures required by trait proxies, as will be discussed in this section.

Trait Proxies In the proposed framework, a proxy is a lazily generated class that implements a type interface *T* and wraps a "core" class `C`. The proxy is needed to map all (and only) the properties declared by *T* to the specific set of (hard) fields provided by `C`, as well as managing those soft fields not provided by `C` directly. Whenever a type *T* must be injected into an object `o` of class `C`, the proxy class `TraitProxy(T,C)` is created (if not existing) and instantiated. The proxy instance wraps the "core" object `o`, providing the type *T* as well as access to the data structures in the implementation. More precisely, a `TraitProxy(T,C)` is compliant with type inheritance, so the instantiation of a proxy `t` for an object `o` is equivalent to the explicit assertion `o` : T and the implicit assertions `o` : S for any $S \mid S :: T$. Calls to the proxy's accessor methods are transparently delegated to the core object. Notice that should a core object be wrapped by two or more proxies that expose the same property, a unique copy of the slot's value will be maintained, thus avoiding duplications and synchronization issues. This is trivially true for hard fields, while soft fields are kept in a data structure that is unique for each core object, regardless of the number of proxies applied.

Type Encoding In order to manage the dynamic types efficiently, a convenient way to index and compare them is used. The set Θ is a directed, acyclic multiple inheritance hierarchy, equipped with a top element $\top$ and a partial order relation $\leq$ modelling subtype inheritance, i.e. for $s, t \in \Theta$, $s \leq t$ implies that s is a subtype ("inherits") of t. So, an efficient and compact encoding mechanism proposed in [6] is adopted. Each element $T \in \Theta$ is assigned a signature key $k(T)$, a bit set, with the properties that $k(\top) = \emptyset$ and $\forall s, t \in \Theta : s \leq t \Leftrightarrow k(s) \succeq k(t)$. The partial order relation $\succeq$ is, in turn, defined as such: denoting with $k(x)[j]$ the $j-th$ bit in a key, $k(s) \succeq k(t) \Leftrightarrow (\forall j : k(t)[j] = 1 \Rightarrow k(s)[j] = 1)$. Informally speaking, the key of the subtype extends the keys of all its supertypes, since any bit set to 1 in any of the latter must also bet set to 1 in the former. So, for example,

if *Patient*, *DiabeticPx* and *OncologyPx* were the only types in the hierarchy, they could be encoded using `1`, `11` and `101` respectively. This encoding allows to map the complex lattice operations $\wedge$ and $\vee$ in Θ to the much simpler boolean bit-wise operations $\&$ and $|$ on the key bitsets. Moreover, while it is reasonable to assume that Θ is statically known at compile time, the encoding strategy is incremental, so that elements can be added later, for example when a knowledge base is expanded to refine the concepts used by an application.

Core Data Structures The core object itself is an instance of class `C`, augmented with the data structures required to hold references to the current traits, soft fields and field type constraints. These additional structures are fixed and can be generated automatically when `C` is initialized. The required structures serve the following purposes:

- Soft Fields Store: The soft field store is a map-like data structure used to hold the current values of an object's soft fields, based on its current dynamic types. The map may be local to the object or global. In the former case, the property can be used as key; in the latter, a two-level index based on the core object and property is required.
- Trait Store: The trait store contains references to the proxies wrapping a specfic object, indexing them by name. Exposed as a Map, it allows to store and retrieve proxies using an identifier (e.g. the fully qualified name) of the interface they implement. Internally, the trait store sorts the proxies by linearizing the partial order defined over the type set Θ, so that proxies can be iterated from the most specific to the most general (*Top*) and vice versa. The ordering is based on a topological linearization algorithm [9] and exploits the bitstet type encoding for efficient comparisons between pairs of proxies.
- Field Metadata Store: For each field (hard or soft), the Field Metadata Store contains reference to a Trait Field Metadata descriptor. This class is responsible for applying the type constraints on a field's value imposed by the current class and types, as defined by equation 1. Given a field's value `x` and a required field type Y here are four possible combinations to consider:
  1. `x` : `C` $\in (\Gamma \cup \Delta)$ and `Y` $\in (\Gamma \cup \Delta)$ s.t. `x` is valid $\iff$ `C` $:: Y$.
  2. `x` : `C` $\in (\Gamma \cup \Delta)$ and `Y` $\in (\Theta)$ s.t. `x` is valid $\iff$ `x` has trait Y.
  3. `x` : $T \in (\Theta)$ and `Y` $\in (\Gamma \cup \Delta)$ s.t. `x` is valid $\iff \exists K$: `x.getCore()` $: K$ and $K :: Y$.
  4. `x` : $T \in (\Theta)$ and `Y` $\in (\Theta)$ s.t. `x` is valid $\iff C :: Y$ or `x.getCore()` has trait Y.

  Notice that options (2) and (4) may be interpreted either as constraints or axioms, depending on whether 3a or 3b is applied.
  The Metadata Descriptor is also responsible for computing a field's candidate default value, based on the current types. Both operations require to compare the currently available traits, according to the partial order on Θ, so they take advantage of the efficient type encoding. Before a value is tentatively assigned to a field, the Descriptor will validate it: whenever the field's

current value does not satisfy the constraints in 1 or 3a, it becomes invalid and the Descriptor will return the default value (if any) to be set in place of leaving the value unspecified.
Notice that, in case of collections, the validation procedure is applied to each element, but the default value is set only in case no valid value is available.

3.3 Production Rule Integration

Production Rule Systems are a widely adopted technology for the implementation of rule-based decision support systems. Their engines are usually based on the Rete algorithm [13], even if alternatives exist such as TREAT [18] and LEAPS [5]. This paper focuses on Rete-based engines because, to this date and to the best of the authors knowledge, most Production Rule Systems implementations are based on this algorithm. A Production Rule Systems system has three major components: (i) a rule base which is compiled into a data processing network, (ii) a working memory (WM) that contains the facts provided to or inferred by the engine and (iii) the engine itself. A detailed description of Rete engines is beyond the scope of this paper, but a formal model of their behavior can be found in [20]. Here, the main area of interests are the working memory operations, namely the assertion, modification and retraction of a fact. In an object-oriented Rete, the first condition checked by discrimination part of the data processing network is the class of a candidate fact, which must match the type defined in a rule pattern. Proxies allow an object o to match a pattern requiring type T even if that object's class C does not implement T directly. To do so, proxies must be asserted into the same WM of their core fact. However, this might lead to unwanted rule activations, since the WM is populated with additional objects which are not under the control of the client's application. To preserve transparency, WM actions and the underlying engine have been extended, so that the core object and its proxies will be treated as a single entity.

Algorithm 1 Generalized WM Type Assertion

```
Require: o : TraitableBean, t: Trait
  if not isA(o,t) then
      T ← new traitProxy( t, o.implClass )
      τ ← encode( mst(o) )
      o.typeLattice.update( T )
      bind( o, T )
      wm.assert( T, τ(o) )
  end if
```

Assertion Let $[o]$ denote the set containing an object and all its current proxies. To enable assertions of the type $o : T$, a new operation is defined $\texttt{don}(o, T)$. This operation does not have any effect if o already has type T at the time it is

called. Assuming then that $\mathtt{o} : T$ does not hold, this operation is responsible for instantiating a proxy implementing T and wrapping $\mathtt{C}$ (where $\mathtt{C}$ is the implementation class of $\mathtt{o}$). This proxy is added to the core object's internal proxy lattice. To ensure that the current set of proxies form a lattice at any time, a $\top$ and a $\bot$ elements are required: the first time `don` is applied to an object $\mathtt{o}$, `don(o,`*Top*`)` is also invoked implicitly. In general, the set will not have a single bottom element: if necessary, the collection is completed with a mock bottom element whose code $\tau(\mathtt{o})$ is the bitwise or ($\vee$) of the currently assigned types. This element is updated whenever the set changes and is removed if an appropriate bottom element can be found among the available traits. This mock type needs not necessarily correspond to the code of an element of Θ, but it allows for the efficient retrieval of the most specific types [6]. This set, denoted by `mst(o)`, is the minimal set of types T_j such that $\mathtt{o} : T_j$ and $\forall T_j \in \mathtt{mst}(\mathtt{o}) \nexists T_k \in \mathtt{mst}(\mathtt{o})$ such that $T_k :: T_j$. The transitive closure of `mst(o)`, based on the sub-type relation ::, is the set of types $\Theta(\mathtt{o}) \subseteq \Theta$ that can be inferred for $\mathtt{o}$. The code $\tau(\mathtt{o})$ has the property that $\forall T \in \Theta(\mathtt{o}) : \tau(\mathtt{o}) \succeq k(T)$.

During the wrapping process, field constraints and default values are computed and applied, based on the semantics described in section 3.1 and exploiting the internal data structures presented in section 3.2. Notice that this may change the value of some fields, and/or require some nested calls to `don`, in case a dynamic type has to be added to an object referenced by one of $\mathtt{o}$'s fields. Once the data structures have been linked, the newly instantiated proxy is asserted as a fact into the WM. The proxy can potentially match any pattern requiring T or any of its supertypes, which $\mathtt{o}$ itself would not have been able to match. However, it is still possible to create undesired activations: it is sufficient that there exists a type S such that $\mathtt{o} : S$ before adding T and $T :: S$. Any pattern matching S would have already been evaluated by $\mathtt{o}$ or one of its previous proxies, so it should not be considered again. To this end, the propagation context of the new proxy T includes the value of $\tau(\mathtt{o})$ before it was updated with $k(T)$. Any time the propagation reaches a RETE type node S, its type code is compared with $\tau(\mathtt{o})$ and the propagation is blocked in case $\tau(\mathtt{o}) \succeq k(S)$. In this case, $\mathtt{o}\mathtt{::}\mathtt{S}$ used to hold even before adding T, so a new evaluation is not necessary. Let's consider for example the coded set

$$\Theta = \{Top(), Patient(1), DiabeticPx(11), OncologyPx(101), FluPx(1001)\}$$

The set includes one main type (*Patient*) descending from *Top* and three of its subtypes. It is assumed that the WM contains a `Person x` with dynamic types $\Theta(\mathtt{x}) = \{Top, DiabeticPx, OncologyPx\}$. Its closure code $\tau(\mathtt{x})$ is 111, which implies *Patient* since $111 \succeq 1$. At this point, `don(x,`*FluPx*`)` is executed and a new proxy is created and configured. Since it is not the case that $1001 \succeq 101$, the new assertion adds novel information, so the proxy must be propagated through the Rete network. The value $\tau(\mathtt{x}) = 111$ is added as context information, allowing the new proxy to match patterns for *FluPx* but not for *Patient*, since the check $\tau(\mathtt{x}) \succeq k(Patient)$ blocks the propagation in that branch of the Rete.

So far, this method would work assuming that the engine supports only one type check per pattern, i.e. that constraints in alpha and beta RETE nodes can

only involve fields. If types are allowed to participate in constraints, condition such as $Patient[\,this$ `instanceOf` $FluPx\,]$ or $FluPx[\,this$ `instanceOf` $OncologyPx\,]$ might be written. The latter would be matched by the propagation of the new proxy, but the former would not, since *Patient* patterns would be blocked. For this reason, and because of the potential changes to the object's field values during the dynamic type injection, the core object `o` must be updated after asserting the new proxy T.

Modification/Update A working memory update is performed to notify the engine that a fact has changed in some way: the engine, in turn, will reevaluate any potentially matching pattern to take the new value(s) into account. To ensure transparency, updates are redefined to operate on [`o`], regardless of whether they are invoked on `o` directly or one of its proxies. The goal is to reevaluate all possibly matching patterns exactly once. First of all, `o` is updated, automatically covering patterns expressed in terms of class `C`, any of its superclasses and any interface that `C` implements statically. Second, the set of dynamic proxies is considered, to reach patterns matching types acquired dynamically at runtime. In particular, only proxies $T_j^{mst} \in$ `mst(o)` are considered. By definition, the type information carried by any other proxy would be implied by at least one element of `mst(o)`. Even then, however, patterns matching a common ancestor of two or more elements of `mst(o)` would potentially be evaluated more than once. Similarly to what was done for assertions, the updates are propagated with a "veto" bit code ν that summarizes the types which have already been covered. Its value is initialized with the mask of all the interfaces implemented by `C` statically. The elements in `mst(o)` are then updated sequentially, passing the veto bitset as a context parameter: after each update, the veto code is bitwise or-ed with the mask of the last propagated proxy. Like in the case of assertions, during the propagation of T_j^{mst}, no node requiring type S (with $T_j^{mst} :: S$) will be (re)evaluated if $\nu \succeq k(S)$. Notice that during the updates triggered by `don` operations, the set `mst(o)` is computed without considering the new type. The algorithm for updates is outlined in Algorithm 2.

Algorithm 2 Generalized WM Update

```
Require: o : TraitableBean
  mst ← o.getMostSpecificTypes()
  ν ← getVetoForClass( o.class )
  for all TraitProxy T ∈ mst do
      update(T, ν)
      ν ← ν ∨ T.getTypeCode()
  end for
```

After the propagation of *FluPx*, the `Person x` is updated. The set `mst(x)` prior to the addition of *FluPx* is `mst(x)` $= \{DiabeticPx, OncologyPx\}$. Since `Person` does not implement any interface, $\nu = \emptyset$. *DiabeticPx* is updated first,

potentially matching pattern types *Top*, *Patient* and *DiabeticPx*. *OncologyPx* is updated next, with $\nu = \emptyset \vee 11$: because of the veto, only patterns for *OncologyPx* can be matched. Nodes requiring *FluPx* have already been evaluated when the specific proxy was inserted. On a normal update, not triggered by a `don,mst(x)` would also include *FluPx*, which would be updated using the veto mask $\nu = 111$, again preventing it from matching *Top* and *Patient*.

Retraction In presence of proxies, retraction can be defined in multiple ways. The default retract operation is redefined to operate on $[\mathtt{o}] : \mathtt{retract}(\mathtt{y} \in [\mathtt{o}]) := \forall \mathtt{x} \in [\mathtt{o}] : \mathtt{retract}(\mathtt{x})$. This ensures that a fact can be retracted regardless of whether it is selected directly or through one of its proxies; at the same time, no proxy will remain in the WM after its core object has been retracted. To allow a user to retract dynamic type assertions such as $\mathtt{o} : T$ a different operation is provided, called `shed`. As the dual counterpart of `don`, `shed(o,`T`)` requires an object `o` and a type T to be executed. Conceptually, it removes any assertion of the type $\mathtt{o} : S \mid S :: T$; concretely, it retracts any proxy of `o` that implements or inherits T. By definition, `shed(o,`T`)` does not require `o` to have type T directly, and will have no effect if `o` does not have any subtype of T. If one or more types are actually removed, however, `[o]` will be updated so that rules triggered by the negation of T can be reevaluated. Notice that the negation of $\mathtt{o} : T$ is assumed to be weak. Production rule engines do not usually support strong negation and rely on negation as failure, so the ability to explicitly state that `o` does not have type T will be the subject of future works.

3.4 Drools-specific implementation

An implementation of the proposed framework has been included in the open source Production Rule Systems engine Drools [21] starting from version 5.6 and above. The implementation builds on top of the engine's features and language and is optimized to offer a trade-off between features and performance.

Language extensions Drools is object-oriented and supports both class and interface-based patterns, even in its native version. From a rule language perspective, the first major extension that is proposed is the introduction of `don` and `shed` as working memory actions, with the behavior described in section 3.3. The second major extension involves the definition of the data models to use in the rules. Drools provides a compact domain specific language for the declaration of java beans as a part of DRL, its native rule language. This language fragment can be described by the following grammar:

⟨*S*⟩ ::= ⟨*Declaration*⟩+
⟨*Declaration*⟩ ::= 'declare' ⟨*JClass*⟩ ('extends' ⟨*JClass*⟩)? ⟨*Body*⟩ 'end'
⟨*Body*⟩ ::= ⟨*JAnnotation*⟩* ⟨*FieldDeclaration*⟩*
⟨*FieldDeclaration*⟩ ::= ⟨*JField*⟩ ':' ⟨*JClass*⟩ ('=' ⟨*JLiteralExpr*⟩)? ⟨*JAnnotation*⟩*

Elements starting with "J" must be compatible with Java class names, annotations, field names and expressions, in order of appearance, and are not further

specified. Classes defined using the `declare` syntax are compiled directly into Java bytecode using ASM [15] and loaded to be used in the rule base runtime. The code compiler will generate fields, constructor(s), accessors, equality/hashCode and toString methods automatically. If present, literal expressions will be used as default values in the constructors, in case no explicit value is provided.

In order to generate traitable beans, it is sufficient to add the annotation `@Traitable` to the class declaration: the code compiler will add the required internal data structures. To generate trait interfaces, instead, a minor extension to the grammar was necessary to add support for multiple inheritance:

⟨*Declaration*⟩ ::= 'declare' ⟨*JClass*⟩ ('extends' ⟨*JClass*⟩)? ⟨*Body*⟩ 'end'
| 'declare trait' ⟨*JClass*⟩ ('extends' ⟨*JClass*⟩)* ⟨*Body*⟩ 'end'

Existing or generated trait interfaces will be annotated with `@Trait`. Notice that while the interfaces are generated at compile time, to allow them to be used in the rules, the trait proxy classes are generated lazily at runtime, the first time that a particular type/class combination is required. The annotations `@Traitable` and `@Trait` are mandatory, but they can be used with preexisting Java classes and interfaces: in that case, the engine will provide the additional data structures at runtime. Both annotations can specify an additional boolean parameter, `false` by default. Unless `@Traitable(logical=true)` is specified, the field type management feature will not be enabled for that core class. While more efficient, it restricts core objects and trait interfaces to always declare the same type for the same property, trivially satisfying (1) or an exception will be thrown. The restriction is necessary since, without field type management, no attempt to cast or convert between objects and proxies can be attempted. Enabling or disabling `@Trait(logical=true)`, instead, allows to differentiate between (3a) and (3b) in case a field's filler object needs a specific type which it does not currently have. Notice that this flag is meaningful only if the core class providing the field is marked as `@Traitable(logical=true)`, regardless of whether the field is hard or soft.

Additional features In addition to the core framework, a number of additional enhancements are provided to facilitate rule authoring and execution.

- Use maps as core objects. Any class implementing `java.util.Map` can be marked as `@Traitable` and used in `don` operations. All fields will be considered soft, but the values will be stored and extracted from the core map. This feature provides a convenient way to mix dynamic data structured with type safety.
- `isA` operator. Type checks can be used both in patterns and constraints, so that rules such as `X( this isA Y )` or `X( field isA Z )` can be written. `isA` generalizes `intanceof` since it considers dynamic types. Its implementation leverages the type codes for efficiency.
- Truth Maintenance. Drools supports "justified" chained assertions [11], which are automatically retracted once the premises supporting them are no longer valid. A "justified" variant of `don` is provided, which automatically

retracts the generated proxies once the rule activation(s) that supported them are no longer valid. Notice that, unlike `shed`, this kind of automated retractions will only involve individual, specific proxies.
- Property Reactivity. Property Reactivity is a configurable form of refraction that was introduced in [19]. On updates, traits would attempt to reevaluate any pattern matching an object's types, regardless of whether they have been acquired statically or dynamically. Property reactivity, then, guarantees that a pattern will be reevaluated if and only if it constrains at least one field that was modified during the update. Unlike the original version, property reactivity has been extended to take the set of dynamic types into account, allowing `isA` to be used with it.

4 Extended Example

Inference and decision rules based on the classification of individuals are pervasive in most business domains. In healthcare, specifically, computerized decision support (CDS) is one of the main areas of application of reasoning technologies. One application of CDS is to check whether a patient has met certain goals and/or schedules, to issue recommendations (re. alerts) with the intent to maintain (re. achieve) compliance to a given care plan. The rules are applied to data provided using standard interchange models, such as the HL7 Reference Information Model (RIM, [28]) or the Fast Interoperable Healthcare Resources (FHIR, [2]). These models define a relatively small number of generic classes, which are meant to be constrained semantically based on the instance data to be represented. For example, a class called `Condition` may be used to model disorders and other ailments, which may or may be not actual illnesses, and may or may not have a confirmed diagnosis. The actual type of Condition would be specified using (medical) terminologies such as SNOMED-CT [16], and the type would imply additional constraints, such as the admissible progression stages, body sites, and symptoms.

For example, a (simplified) diabetes management rule may state that *"Diabetic Patients with a poorly managed blood sugar concentration should be seen every 2-3 months"*. Rules of this kind predicates on non-rigid types such as 'patient', '(well managed) diabetes' and 'blood sugar', which apply to classes such as 'person', 'condition' and 'laboratory test result', respectively. In order to execute this business logic in a rule engine, one would have to rewrite the constraints in a form that is more operational, such as:

```
rule "Diabetes Recommendation" when
  $per: Person($name : name, addr: address)
  $pat: Patient(person == $per)
  // confirmed diagnosis of active diabetes
  $dia: Condition(subj == $pat, code == "sct:73211009", status == A)
  // uncontrolled blood sugar, assessed via Hemoglobin A1C test
  $a1c: Observation(subj == $per, code == "lnc:74246-0", value > 7.0)
then
 // Schedule next appointment 2-3 months from now
end
```

Rules like this mix business and data concepts, and are less understandable than the original guideline. Moreover, they tend to overfit the intent business logic. There are several ways to infer that a Patient "has diabetes", not all of which have enough context to instantiate an object of type Condition consistently. Likewise, a specific lab test (with a threshold that is subject to variation) is not the only way to infer that a person's blood sugar is "uncontrolled". To mitigate these issues, one could make the intermediary inferences explicit, leveraging domain (and rule-) specific facts:

```
declare Diabetic
  subj : Patient
end

declare HasPoorlyControlledBloodSugar
  subj : Patient
end

rule "Diabetic" when
  $pat: Patient()
  $dia: Condition(subj == $pat, code == "sct:73211009", status == A)
then
  insertLogical( new Diabetic($pat) );
end

rule "Blood Sugar" when
  $pat: Patient()
  $a1c: Observation(subj == $at, code == "lnc:74246-0",
  value#Quantity > 7.0%) // 'value' is variant, so a cast is needed
then
  insertLogical( new HasPoorlyControlledBloodSugar($pat) );
end

rule "Diabetes Recommendation" when
  $per: Person($name : name, addr: address)
  $pat: Patient(person == $per)
  $dia: Diabetic(subj == $pat)
  $sug: HasPoorlyControlledBloodSugar(subj == $pat)
then
 // Schedule next appointment
end
```

This wording emphasizes the domain concepts, but the knowledge engineer writing the rules has to maintain the consistency between the domain objects and the helper facts, using technical features of rule engines such as truth maintenance and/or refraction. For example, the 'HasPoorlyControlledBloodSugar' fact holds only as long as the Hemoglobin value remains above the threshold. Moreover, joins are still required to match the marker facts with the corresponding core objects, increasing the complexity of the underlying RETE network. Using traits, instead, type markers can still represent domain concepts, but the integrity of the working memory is handled by the engine rather than the user:

```
declare trait Patient
  mrn: String // medical record number - soft field
```

```
end
declare trait Diabetic extends Patient end
declare trait Diabetes end
declare trait HemoglobinA1CTest
  value: Quantity // type restriction
end
declare trait HasPoorlyControlledBloodSugar extends Patient end

rule "Patient" when
  $per: Person()
  exists Condition(subj == $per)
then don($per, Patient.class).with( mrn = getMRN($per$) ); end

rule "Diabetes" when
  $con: Condition(code == "sct:73211009", status == A)
then don($con, Diabetes.class); end

rule "HgbA1c" when
  $a1c: Observation(code == "lnc:74246-0")
then don($a1c, HemoglobinA1CTest.class); end

rule "Poorly managed diabetes" when
  $pat: Patient()
  $dia: Diabetes(subj == $pat)
  $a1c: HemoglobinA1CTest(subj == $pat, value > 7.0% ) // no cast
then
  $don($pat,{Diabetic.class, HasPoorlyControlledBloodSugar.class});
end

rule "Diabetes Recommendation" when
  $per: Diabetic( $name : name, $addr: address, $mrn : mrn,
    this isA HasPoorlyControlledBloodSugar.class)
then
 // Schedule next appointment
end
```

This formalization, which is one of many possible, demonstrates some aspects of the proposed approach. The rule base is larger, but captures a deeper level of domain knowledge, leveraging concepts with domain semantics rather than data semantics. It allows for a better separation of concerns, decoupling inference and classification from decision making. Traiting allows to introduce constraints in the forms of convenient types and casts. An object can have multiple traits within the same type hierarchy, so `Patient(this isA Diabetic.class)` and `Diabetic()` are equivalen to `Person(this isA Diabetic.class)`: in particular, a rule would activate only once for an instance of Person that has both the Patient and the Diabetic trait since the latter implies and thus masks the former. From a technical perspective, type checks are enforced by means of alpha node constraints, rather than beta node ones, resulting in a simpler RETE network. Finally, truth maintenance is handled by the engine automatically.

5 Benchmarks and Results

The use of traits increases the expressivity of the rule language, and consequently the complexity of the rule engine. Even if this complexity is hidden from the knowledge engineers and subject matter experts, it has implications in terms of performance. To prove that this additional cost is balanced by the ability to construct simpler rules, we have defined a suite of synthetic benchmarks that mimic the basic trait usage patterns. Japex [1] is used to write the java-based micro-benchmarks and run them with appropriate warmups and repetitions. All the test suites ran on a desktop machine with an Intel Core i5 processor, 8GB of RAM, Microsoft Windows 7 64bit OS, JDK 7, Drools 5.6 and Japex 1.1.3. The benchmarks compare the use of traits against marker facts emulating the same logic provided by traits, measuring the overall cost of traits execution against the join cost of the Rete algorithm [3]. The first test evaluates the scalability of the system as the number of `isA` constraints increases, with a very large number of dynamic types. The results in Table 1 show the trait version was faster and scaled better with the number of dynamic types. This implies that the more types are possibly matched by a fact, the more convenient it is to use traits instead of helper objects.

Table 1: speed-up with increasing number of isA in μs

Iteration	Native	Trait	Native/Trait
100	1.824	1.539	1.185
500	5.109	3.579	1.427
1000	13.535	7.167	1.888

The second benchmark, Figure 1, focuses on a single insert/join replaced by a don/isA, monitoring its execution time as the pattern gets applied multiple times to multiple facts. Just In Time (JIT) compilation causes the initial speed increase, with traits taking longer due to the proxy runtime generation. As iterations increase, the trait version becomes asymptotically more efficient, possibly due to the simpler structure of the Rete. The third benchmark, Figure 2, was designed to see how adding fields to traits and using them in constraints would affect the overall performance. In this benchmark, a single trait was declared in each test with a different number of fields, mixing soft, hard and hidden ones. The results show the execution time was not affected by hidden fields, but increased with the number of soft and hard fields. In all benchmarks, the measurements have been reproducible and consistent, as shown by the low value of the standard deviation. The small differences in the value of the average, computed using different strategies, confirm the absence of outliers in the measurements.

Fig. 1: Rule execution time during the initial transition : trait vs native

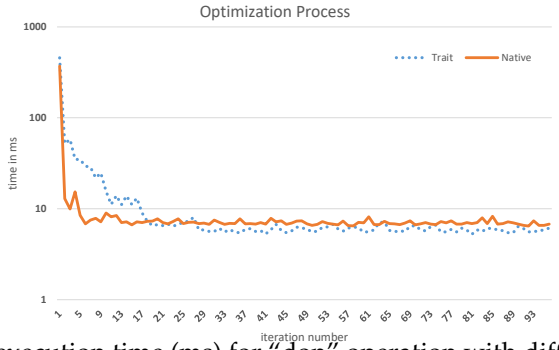
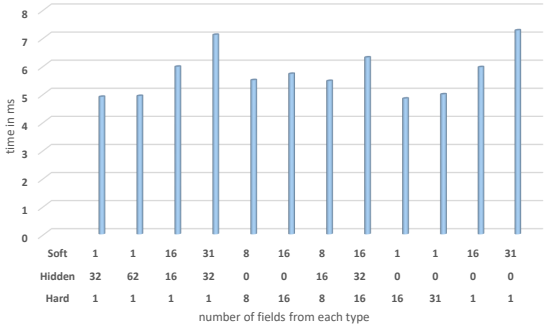


Fig. 2: Average execution time (ms) for "don" operation with different types and number of fields



6 Conclusions

We have proposed an extension for a on object oriented production rule system, based on the concept of dynamically typed facts. The approach combines the flexibility of dynamic data structures while enabling the benefits of type safety. To this end, we have extended the Rete engine with the addition of two new working memory actions, `don` and `shed`, and redefined the existing ones (`assert`, `retract` and `update`) to work with the new specification.

The extensions are conservative and do not impact the engine unless used explicitly. Moreover, they provide a good compromise between memory consumption, execution time, data integrity, declarativeness and expressivity. As shown by the benchmarks, the use of traits is not detrimental to performance except for the simplest cases, while there exists cases where it has actually been proven to be beneficial. In general, any time that traits allow to simplify the structure of a rule, some kind of performance gain is to be expected. A precise quantification of this gain is hard to predict in practice, since it heavily depends on the use cases and will be the subject of further studies. The only important aspect to remember is that trait proxy classes are generated lazily the first time that they are needed.

From a knowledge represenaion perspective, the proposed framework allows rules to be written against clean conceptual domain models, implemented

using interfaces, rather than concrete information models. This approach can be considered a good practice in general: it allows the decoupling of the rules from the facts they are supposed to match and increases their portability between different systems, which is a well known limit in areas where automated decision making plays an important role.

From a theoretical perspective, there should be further investigate the integration of production rules with ontologies and other formalism for the definition of concepual models. Preliminary works, not discussed in this paper, show that trait-oriented domain models (as well as companion information models) could be derived from ontologies automatically, and will be the subject of future publications. On top of this, the potential role of traits in bridging some of the differences between rule-based and description logic reasoning will be explored.

References

1. Japex (2007), https://japex.java.net/docs/manual.html
2. FHIR (2012), http://www.hl7.org/implement/standards/fhir/index.html
3. Albert, L., Fages, F.: Average case complexity analysis of the rete multi-pattern match algorithm. In: Proceedings of the 15th International Colloquium on Automata, Languages and Programming. pp. 18–37. ICALP '88, Springer-Verlag, London, UK, UK (1988)
4. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York, NY, USA (2003)
5. Batory, D.: The leaps algorithm. Tech. rep., Austin, TX, USA (1994)
6. van Bommel, M.F., Wang, P.: Encoding multiple inheritance hierarchies for lattice operations. Data Knowl. Eng. **50**(2), 175–194 (Aug 2004)
7. Cardelli, L.: A semantics of multiple inheritance. In: Proc. of the international symposium on Semantics of data types. pp. 51–67. Springer-Verlag New York, Inc., New York, NY, USA (1984)
8. Chugh, R., Rondon, P.M., Jhala, R.: Nested refinements: A logic for duck typing. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 231–244. POPL '12, ACM, New York, NY, USA (2012)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 2 edn. (2001)
10. Curry, G., Baer, L., Lipkie, D., Lee, B.: Traits: An approach to multiple-inheritance subclassing. ACM SIGOA Newsletter **3**(1-2), 1–9 (Jun 1982)
11. Doyle, J.: A truth maintenance system. Artif. Intell. **12**(3), 231–272 (1979)
12. Fikes, R., Kehler, T.: The role of frame-based representation in reasoning. Commun. ACM **28**(9), 904–920 (Sep 1985)
13. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. **19**(1), 17–37 (1982)
14. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. J. ACM **42**(4), 741–843 (Jul 1995)
15. Kuleshov, E.: Using the asm framework to implement common java bytecode transformation patterns (2007)

16. Lee, D., Cornet, R., Lau, F., de Keizer, N.: A survey of snomed ct implementations. Journal of Biomedical Informatics **46**(1), 87 – 96 (2013)
17. Lipkie, D.E., Evans, S.R., Newlin, J.K., Weissman, R.L.: Star graphics: An object-oriented implementation. SIGGRAPH Comput. Graph. **16**(3), 115–124 (Jul 1982)
18. Miranker, D.P.: Treat: A better match algorithm for ai production systems. In: Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1. pp. 42–47. AAAI'87, AAAI Press (1987)
19. Proctor, M., Fusco, M., Sottara, D.: Extending an object-oriented rete network with fine-grained reactivity to property modifications. In: Morgenstern, L., Stefaneas, P.S., Lévy, F., Wyner, A., Paschke, A. (eds.) RuleML. Lecture Notes in Computer Science, vol. 8035, pp. 173–187. Springer (2013)
20. Projet, S., Formalisation, A., Cirstea, H., Kirchner, C., Moossen, M., etienne Moreau, P., Lorraine, I., Historique, I.R.: Production systems and rete
21. Red Hat: Drools (2019), www.drools.org
22. Rose, J.: Jsr 292: Supporting dynamically typed languages on the javatm platform (2011)
23. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: In Proc. European Conference on Object-Oriented Programming. pp. 248–274. Springer (2003)
24. Stevenson, G., Dobson, S.: Sapphire: Generating java runtime artefacts from owl ontologies. In: Salinesi, C., Pastor, O. (eds.) CAiSE Workshops. Lecture Notes in Business Information Processing, vol. 83, pp. 425–436. Springer (2011)
25. Syme, D., Battocchi, K., Takeda, K., Malayeri, D., Fisher, J., Liu, T., Mcnamara, B., Quirk, D., Taveggia, M., Chae, W., Matsveyeu, U., Petricek, T.: Strongly-typed language support for internet- scale information sources (2012)
26. Ungar, D., Smith, R.B.: Self: The power of simplicity. pp. 227–242 (1987)
27. Yang, G., Kifer, M.: Inheritance in rule-based frame systems: Semantics and inference. **4244**, 79–135 (2006)
28. Zhang, Y.F., Tian, Y., Zhou, T.S., Araki, K., Li, J.S.: Integrating hl7 rim and ontology for unified knowledge and data representation in clinical decision support systems. Comput. Methods Prog. Biomed. **123**(C), 94–108 (Jan 2016)