

UNIVERSITEIT VAN AMSTERDAM

MASTERS PROJECT

Representation Mismatch Reduction for Development in Rules-Based Business Engines

Author:

Paul SPENCER

Supervisor:

Dr. Clemens GRELCK

*A thesis submitted in fulfillment of the requirements
for the degree of Master of Software Engineering*

in the

**Graduate School of Informatics
Faculty of Science**

August 3, 2021



UNIVERSITY OF AMSTERDAM

Declaration of Authorship

I, Paul SPENCER, declare that this thesis titled, “Representation Mismatch Reduction for Development in Rules-Based Business Engines” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this thesis is entirely my work.
- I have acknowledged all of the main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UNIVERSITEIT VAN AMSTERDAM

Abstract

Graduate School of Informatics

Faculty of Science

Master of Software Engineering

Representation Mismatch Reduction for Development in Rules-Based Business Engines

by Paul SPENCER

Context: Declarative rules engine languages, such as Drools, can become difficult to reason about when there are many rules.

Objective: This project investigates how different projections of the code can ease the comprehensibility of the code.

Method: We created an implementation of the Drools language using the MPS language workbench and made innovative projections of large ASTs.

Results:

Keywords: projectional editing; Rules Engines; MPS; Drools

Paper type: Research paper

Acknowledgements

We would like to acknowledge the School of The Graduate school of Informatics in the Faculty of Science at The University of Amsterdam for their guidance, specifically Dr. Clemens Grelck, who has been a supportive, understanding, and available academic adviser.

We received inspiration from the Strumenta Languages engineering community. Specifically we would like to thank Federico Tomasetti, who shared with me his model of a rules engine in MPS.

Also we would like to thank Václav Pech from JetBrains for the course he created and the time he spent with me explaining MPS. Further, Sergej Koščejev from JetBrains helped us with specific MPS issues during his Office hours.

Other prolific output that terrifically helped our research and development was the Heavy Meta YouTube series from Kolja Dumann and the dozens of papers and books from Markus Voelter, both currently working at Itemis A.G.

[TODO: add mark proctor if he helps out]

Our greatest thanks go out to Toine Khonraad, an alum of this course, who provided me with moral and monetary support, as well as wisdom and friendship that aided in the completion of this, my fourth attempt at getting this project behind me. Without his constant mantra of simplify, simplify, simplify, we would still be implementing the Drools languages now without having made a single projection.

Contents

Declaration of Authorship	iii
Abstract	v
Acknowledgements	vii
1 Introduction	1
1.1 Problem statement	1
1.2 Research questions	2
1.3 Contributions	3
1.4 Project context	3
1.5 Thesis outline	3
2 Background	5
2.1 RulesEngines	5
2.1.1 What is a rules engine?	5
2.1.2 What is Drools?	7
An explanatory example	10
2.2 Projectional Editing	12
2.2.1 What is projectional editing?	12
parser based editing	12
projectional definition	12
How projectional editing works	13
What advantages does projectional editing bring?	14
2.2.2 What are Language Workbenches?	19
2.2.3 What is MPS?	19

3 Method	21
3.0.1 constructing prototypes	21
4 Results	23
5 Discussion	25
5.1 Threats to Validity	25
5.1.1 Construct Validity	25
5.1.2 Internal Validity	25
5.1.3 External Validity	25
5.1.4 Reliability	25
5.1.5 Repeatability vs Reproducibility	25
5.1.6 Method improvement	25
6 Implications to research and practice	27
6.1 Implications to research	27
6.2 Future research directions	27
6.3 Implications to practice	27
7 Conclusion	29
A InterviewTranscripts	31
Bibliography	33

List of Figures

2.1 Drools components.	8
2.2 Drools Inference Loop.	9
2.3 Drools Rule Breakdown.	10
2.4 Projectional editing loop.	20

List of Tables

2.1 Rules Engine products.	7
------------------------------------	---

Chapter 1

Introduction

The limits of my language mean the
limits of my world.

Logico-Tractatus Philosophicus
Ludwig Wittgenstein

1.1 Problem statement

Miller's Law[1] states that an average human can hold in his short-term memory 5-9 objects. This is often an argument for more succinct code. The argument being anything that is not immediately in the developers vision has to be stored in her memory. With it being impractical to reason about code that she cannot recall, then the fewer relevant items to her reasoning that are out of view the easier it is to reason about the code.

Our host organization, Khonraad Software Engineering, a subsidiary of Visma, provides mission-critical services focussed on the automation of workflows at the cross-section of local government and healthcare. Specifically, Khonraad facilitates the mental health care and coercion laws in the Netherlands - WVGZ, WZD, and WTH - which provide agencies the ability to intervene in domestic violence, psychiatric disorders, and illnesses.

Khonraad's system facilitates reporting and communication between municipalities, police, judiciary, lawyers, mental health care, and many social care institutions. The system has 15,000 users and is available 24/7.

Configuration and administration use complex matrices of compliance mechanisms, access user rights and communication settings. The sensitivity of the personal data, being both medical and criminal, means security is of utmost importance. The security against data loss, preventing unlawful disclosure and guaranteeing availability, especially during crisis situations, is crucial. Demonstration of the correctness of the, often changing, configuration is a major concern in the company.

This configuration is done in a business rule system, specifically JBoss Drools.

Drools is a language that shares an unfortunate characteristic with many other rules languages. It is verbose and can contain many rules that can interact with each other without obvious visual connection. As Forgy[2] points out, for production systems in general, "production systems have another property that makes them particularly attractive

for constructing large programs: they do not require the programmer to specify in minute detail exactly how the various parts of the program will interact”. This property leads to very large and hard to reason about collections of implicitly connected rules.

Reasoning over a small number of rules is already surprisingly hard. Our host organization has many rules and, thus, reasoning about them is particularly challenging.

We have observed the difficulty that developers have trying to reason about and edit collections of Drools files. We hypothesize that developers can be presented with different views on their code that will allow them to better understand the code. The problem we wish to solve - how to improve the ability to reason about large collections of Drools rules - we believe, lends itself to the technique of projectional editing. By using projections to improve feedback whilst coding, we believe that this can reduce the representation impedance mismatch that hampers developer’s reasoning.

The problem considered in this thesis is how to present rules to a developer in a way in which she can interact with [TODO: finish this thought]. As is perhaps already obvious it is not our intention to override the will of the language engineers who have spent many years developing this language and it’s ecosystem. The goal of this thesis is to augment the current developer experience.

1.2 Research questions

To reason about a large code base of rules engine code effectively, a different presentation is needed. This presentation should allow a clearer organization whilst remaining interactive. We can formulate the following research questions based on the discussion in the preceding sections.

The research question we wish to answer is:

- **Main research question:** “How can projectional editors and DSLs be combined to address feedback mechanisms for developers in the context of reasoning about rules in a rule-based business engine?”

This question requires knowing if it is possible with current tooling, thus we would like to answer the question:

- **RQ 1:** “What is the current state of language workbenches supporting projectional editing?”

Finally, we specifically would like to know how we can improve the ability to reason about the business rules engine, so we ask the question:

- **RQ 2:** “Which projections can help developers to get appropriate feedback about rules?”

1.3 Contributions

This thesis proposes a code representation of business rules in a concise and readable format that could solve comprehensibility issues resulting from large code bases of business rules. The implementation behind the approach relies on language engineering and projectional editing. An implementation has been developed as a stand alone opensource solution on a limited demonstration version of Drools. The underlying Drools implementation can be used as a base language for model to model generation by the wider MPS ecosystem.

1.4 Project context

This investigation was hosted by Khonraad Software Engineering, a subsidiary of Visma. Khonraad provides mission-critical services focussed on the automation of workflows at the cross-section of local government and healthcare. Specifically, Khonraad facilitates the mental health care and coercion laws in the Netherlands - WVGZ, WZD, and WTH - which provide agencies the ability to intervene in domestic violence, psychiatric disorders, and illnesses.

Khonraad's system facilitates reporting and communication between municipalities, police, judiciary, lawyers, mental health care, and many social care institutions. The system has 15,000 users and is available 24/7.

Configuration and administration use complex matrices of compliance mechanisms, access user rights and communication settings. The sensitivity of the personal data, being both medical and criminal, means security is of utmost importance. The security against data loss, preventing unlawful disclosure and guaranteeing availability, especially during crisis situations, is crucial. Demonstration of the correctness of the, often changing, configuration is a major concern in the company.

This work environment allows us to work on an existing project, where the tangible success will have an impact on the lives of those in critical need. Khonraad has its own implementations in the Drools language, that have evolved over the iterations of the laws. The evolution of the code base over the years means that the real-life issues we came across are not just thought experiments.

1.5 Thesis outline

We start in chapter 2 with the required background information on projectional editing and rules engines. In chapter 3 we present the research questions. Further, the chapter describes the protocol that we use for search strategy, selecting our studies, extracting data from them, and synthesizing the results. Chapter 4 presents the results of our synthesis of data from the primary studies. This is followed, in chapter 5, by a discussion of both the validity of the work and the implications of the findings. We discuss the implications of this study in chapter 6. Finally, the conclusions are presented in chapter 7.

Chapter 2

Background

This chapter gives the background information required on rules engines and projectional editing. It presents the specific case of rules engine that we will be using for our investigation: Drools. Further, it briefly examines the base tool type for creating Domain-specific languages: Language work benches. Finally, it presents the specific projectional editing tool we will be using: JetBrains MPS.

2.1 RulesEngines

2.1.1 What is a rules engine?

In this section we will describe what a rules engine is and a little of its history.

The Aristotelian doctrine of essentialism declares that a thing has properties that are essential and properties that are accidental. If one takes away accidental properties, then the thing remains the thing. If one takes away essential properties, the thing is no longer the thing. If the thing is a business application, then its essential properties are its business rules.

Simply put, business rules are the principles or regulations by which an organization carries out the tasks needed to achieve their goals. When properly defined these rules can be encoded into statements that defines or constrains some aspect of the business organizational behaviour. A rule consists of a condition and an action. When the condition is satisfied then the action is performed. More formally, business rules can be seen as the implication in the basic logical principle of Modus Ponens.

When described like this, one could me forgiven for thinking is this not just an if-then logic that is frequently used in traditional programming. One would not be wrong, however in traditional programming, representing all the combinatorial outcomes can become complex. In the typical application architecture, rules are distributed in the source code or database. Each additional rule leads to more fragility.

Documentation describing these rules may be found in the design documentation or user manuals. However, as applications evolve documentation gets out of sync with codebase. Once this desynchronization occurs, to know what the rules that govern the application, one has to navigate the codebase and decode the rules from their, often scattered, locations.

A rules engine is also known as a Business Rules Engine, a Business Rules Management System or a Production Rules System. The goal of a rules engine is the abstraction of business rules into encoded and packaged logic that defines the tasks of an organization with the accompanying tools that evaluate and execute these rules. Simply put, they are where we evaluate our rules. Rules engines match rules against facts and infer conclusions. If we return to the Modus Ponens comparison:

$$\begin{array}{l} p \\ p \rightarrow q \\ \hline \therefore q \end{array}$$

If the premise p holds. And the implication $p \rightarrow q$ holds then the conclusion q holds. In terms of a rule engine and business rules this could be seen as:

1. the rules engine gathers the data for the premise: p
2. it examines the business rules as the implications: $p \rightarrow q$
3. it executes the conclusion: q

Rules engines follow the recognize-act cycle. First the match, i.e. are there any rules with a true condition, Next the do conflict resolution, to pick the most relevant of the matching rules, finally they act, which is to perform the actions described in the rule. If no items are matched then the cycle is terminated, otherwise the first step is returned to.

Rules Engines are declarative, focussing on the what of the rules not the the how of the execution. Date[3] describes rules engine as to “specify business process declaratively, via business rules and get the system to compile those rules in to the necessary procedural (and executable) code.” Fowler[4] describes rules engine as follows: “ ... providing an alternative computational model. Instead of the usual imperative model, which consists of commands in sequence with conditionals and loops, a rules engine is based on a Production Rule System. This is a set of production rules, each of which has a condition and an action ...”.

Rule engines arose from the expert systems of the late 70s and early 80s. Expert systems initially had three main techniques for knowledge representation: Rules, frames and logic[5]. "The granddaddy" of the expert systems, MYCIN, relied heavily on rules based knowledge representation[6], rather than long inference chains. MYCIN was used to identify bacteria and recommend antibiotic prescriptions. MYCIN and its progenitor, DEN-DRAL, spawned a whole family of Clinical Decision Support Systems that pushed the rules engine technology until the early 1980's. Research into rules engines died out in the 1980s as it fell out of fashion.

Early in their existence, the rules engines hit a limiting factor because the matching algorithms they used suffered from the utility problem, i.e. the match cost increased linearly with the number of rules being examined/ This problem was solved by Charles Forgy's efficient pattern matching Rete algorithm[2], and its successors. This algorithm works by modelling the rules as a network of nodes where each node type works as a filter. A fact will be filtered through this network. The pre-calculation of this network is what provides the performance characteristics.

The first popular rules engine was Office Production System from 1976. In 1981 OPS5 added the Rete algorithm. CLIPS in .. JESS Drools

Product		Developer	licence type
CLIPS	[7]	NASA	open source
Drools	[8]	JBoss/RedHat	open source
BizTalk Business Rule Engine	[9]	Microsoft	proprietary
WebSphere ILOG JRules	[10]	IBM	proprietary
OpenRules	[11]	OpenRules	open source

TABLE 2.1: Rules Engine products.

In general, rules engines are forward chaining. This means to test if [TODO: Explain forward chaining with logic symbols]

[TODO: ADD MORE HISTORY HERE]

Moving forward to current times, there are a few rules engines currently in use. Some of the more commonly used ones are shown in table 2.1

Some of the advantages of using a rules engine include:

- The separation of knowledge from it's implementation logic
- Business logic can be externalized
- Rules can be human readable

Rules that represent policies are easily communicated and understood. Rules retain a higher level of independence than conventional programming languages. Rules separate knowledge from its implementation logic. Rules can be changed without changing source code; thus, there is no need to recompile the application's code. Cost of production and maintenance decreases.

In summary a rules engine, is the executor of a rules based program, consisting of discreet declarative rules which model a part of the business domain.

2.1.2 What is Drools?

JBoss Rules, or as it is more commonly known, Drools, is the leading opensource rules engine written in Java. In this paper when we use the name "Drool" we are referring to the "Drools Expert" which is the rule engine module of the Drools Suite. Drools started in 2001, but rose to prominence with it's 2005 2.0 release. It is an advanced inference engine using an enhanced version of the Rete algorithm, called ReteOO[12], adapted to an object-oriented interface specifically for Java. Designed to accept pluggable language implementations, it can also work with Python and .Net. It is considered one of the most developed and supported rules platforms.

For rules to be executed there are 4 major components as demonstrated in figure 2.1. The production memory contains the rules. This will not change during an analysis session. The rules are the focus of this thesis and therefore we will delve into much more detail later on these.

In Forgy's[2] overview of a rete algorithm, the following steps occur.

1. Match : Evaluate the LHSs of the productions to determine which are satisfied given the current contents of working memory
2. Conflict resolution : Select one production with a satisfied LHS; if no productions have satisfied LHSs, halt the interpreter
3. Act : Perform the actions in the RHS of the selected production
4. Re-evaluate : Go To 1

Figure 2.2 show more detail of how these components interact within Drools to infer a conclusion. First a fact or facts are asserted in the working memory. The working memory contains the current state of the facts. This triggers the inference engine. The pattern matcher, using the aforementioned ReteOO algorithm will determine examine the working memory and a representation of the rules from the production memory to determine which rules are true. Matching rules will be placed on the agenda. It can be the case that many rules are concurrently true for the same fact assertion. These rules are in conflict. A conflict resolution strategy will decide which rule will fire in which order from the agenda. The first rule on the agenda will fire. If the rule modifies, retracts or asserts a fact, then the inference loop begins again. If a rule specifies to halt or there are no matching rules left on the agenda, we have inferred our conclusion.

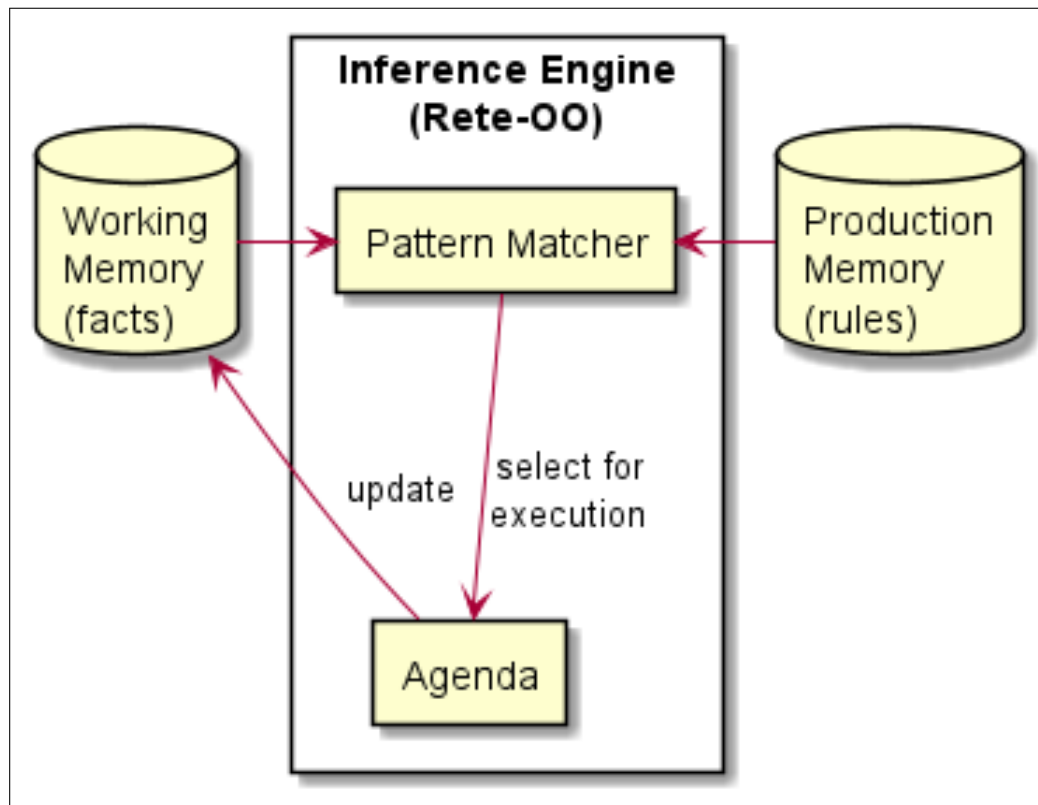


FIGURE 2.1: Drools components.

The component we will be focussing on in this paper is the rules. Rules are stored in a rules file, a text file, typically with a `drl` extension. During execution the rules do not change and are stored in production memory. For the sake of this paper we will skip `past`, `package`, `import`, `global`, `declare`, `function` and `query`, which are also stored in the rule file. We will examine the anatomy of a rule.

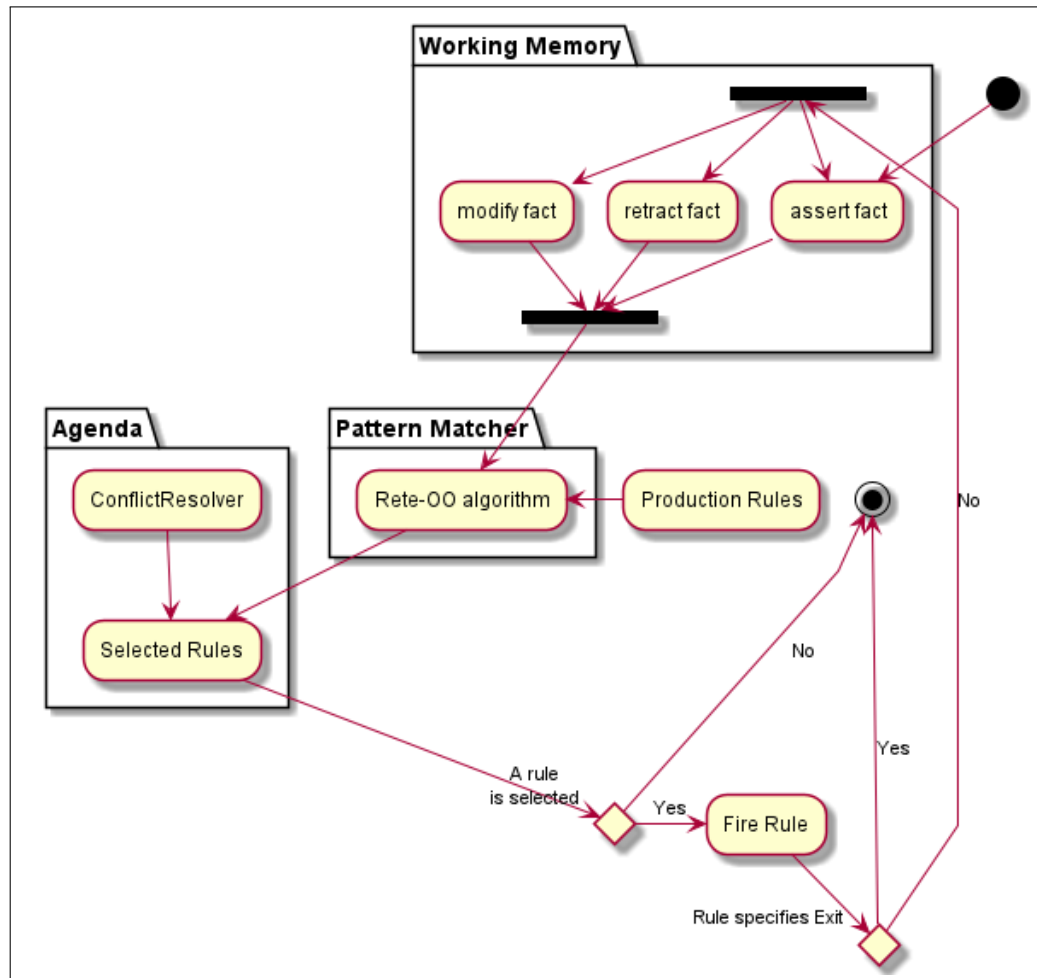


FIGURE 2.2: Drools Inference Loop.

A rule is made of 3 parts: attributes; conditions; and consequences. Attributes are an optional hints to the inference engine as to how the rule should be examined. The conditional, when, or left hand side (LHS) of the rule statement is a block of conditions that have to in aggregate return true for the asserted fact in order to be considered to be placed on the agenda. The actions, consequences, then, or Right hand side (RHS) of the rule statement contains actions to be executed, should the rule be filtered

The LHS is a predicate statement, made up of a number of patterns. variables can be bound to facts that match these patterns for use later in the LHS or for updating the working memory on the RHS. the patterns are used to evaluate against the working memory. The pattern match against the existence of facts. Patterns can also match against conditions of the properties of facts. Connectives such as not, and, and or can be applied to the patterns. The patterns apply to individual Facts rather than the group, thus can be seen as first order predicates.

There are some more advanced features in the LHS, but for this paper, these are the features we will be looking at.

Whilst the RHS can contain arbitrary code to be executed when a rule is fired, it's main purpose is to adjust the state of truth in the working memory. One can insert, modify, and retract facts in the working memory. modifying and retracting facts, must be done on fact


```
29     rule "Corrupt the Honest"
30     when
31         $p : Politician( honest == true )
32         exists( Hope() )
33     then
34         System.out.println( "I'm an evil corporation and I have
35                             corrupted " + $p.getName() );
36         modify( $p ) {
37             setHonest( false )
38         }
39     end
```

LISTING 2.1: Example Drools file.

Listing 2.1 gives the Drools engine instructions on what actions to take when something changes in the working memory. What this toy example does is reacts to when an honest politician is added to the working memory, prints a message celebrating the existence of said politician, corrupts her, gloats in a message and then prints a message of despair. The code in listing 2.1 does the following:

1. on line 1 the package statement identifies the rule file
2. on lines 3 and 4 the import statements describes which facts can be used
3. the “We have an honest Politician” rule on line 6 does the following:
 - (a) using salience on line 7 it sets that this rule is to be run before rules with a lower salience
 - (b) on line 10 it checks the working memory for Politician facts with the honest property equal to true
 - (c) on line 12, if found then Hope facts will be inserted into the working memory
4. the “Hope Lives” rule on line 15 does the following:
 - (a) line 18 check if any Hope facts exist
 - (b) on line 20, if found, it prints a message
5. the “Hope is Dead” rule on line 23 does the following:
 - (a) checks if no Hope facts exist on line 25
 - (b) if none are found, on line 27, it prints a message
6. the “Corrupt the Honest” rule on line 30 does the following:
 - (a) line 32 checks for any Politician facts with the honest property equal to true, and sets them to the variable \$p
 - (b) line 33 checks if any Hope facts exist
 - (c) if both hope and politicians are found on line 35 it prints a message including the \$p variables name
 - (d) on line 36 to 38 it modifies the fact in working memory represented by \$p to change it’s honest property

2.2 Projectional Editing

2.2.1 What is projectional editing?

Traditionally programmers write code with text editors, or integrated development environments (IDE), which are basically text editors on steroids. A projectional editor, also known as structured editor or syntax-directed editor, is a different way to write code.

parser based editing

In a traditional, parser-based, development workflow, users use a text editor to represent the program. Because it is text based then the notation of the language is limited to text. A grammar is a definition of the formal syntactical rules, or concrete syntax, of a programming language. The lexer and parser are derived from the grammar. The text is passed into a text buffer where first a lexer will turn the text into tokens. A parser will validate that these tokens, the words of the language, are syntactically correct. If the tokens are not rejected then the parser will construct first a parse, or concrete syntax, tree and from there, an abstract syntax tree (AST).

An AST is a tree structure that represents the semantic meaning of the source code, stripped of all the syntactic details. The parser will do some name resolution to take care that references within the source code are represented in the tree. This turns the tree into a graph.

Compilers use the AST to do subsequent processing, such as linking, transformation, analysis, type checking, etc. Modern IDEs, in the background, also parse the code it is displaying to create an AST in order to offer relevant coding assistance. This assistance is appreciated, because without IDE help learning the concrete syntax of large languages is error prone and exploratory programming is laborious if one has to wait until compilation to discover mistakes.

projectional definition

A projectional editor does not parse any text. In its place, a developer reads and edits the AST through a projected notation. Her editing gestures immediately and directly manipulate the AST within predefined and fixed layouts.

The principle of projectional editing is familiar to those that use visual programming, like Scratch or Blockly, or graphical modelling tools, such as MetaEdit+. These tools do not parse pixels to get to their AST. They project the underlying models/programs in a view and which they store as the model/AST and not as a plain text equivalent in a traditional programming language.

Projectional editing is the generalization of this idea, with the ability to render multiple representation of the program with a wide range of notation styles.

The projection may be textual. It also may be any other notation that can represent the semantic meaning of the code, such as formulas, graphs, or images. Projections are

not just the notation, but also how the user interacts with the projection. In this sense the definition of the projections and the IDE/UI overlap.

How projectional editing works

As shown in figure 2.4, a projectional editor has a model or an AST. It renders a presentation of the model as a projection. The developer performs actions on the projection. every user editing action is directly mapped on to a change in the AST.

To perform the above two things have to be defined by language engineers: The Meta-Model and the editor.

The meta-model, analogous to the abstract syntax, describes the node concepts and connection that can be used to build the, hierarchical structure that is the AST. This hierarchy can have references to nodes in other branches, so, although named a tree, it is actually a graph. The AST will be stored independently of the concrete syntax. The tree is often stored with a database, XML or a proprietary file format.

Rules of the meta-model can further be described through behaviours such as type systems or scoping rules.

Projectional editors avoid the grammars and parsers that serve as the definition of the concrete and abstract syntax in a traditional text based language. Instead of transforming the concrete to the abstract with parsing, the abstract is transformed to the concrete via a projection engine that uses projection rules. Editors are a combination of the projection rules and the gestures or actions that will create change request to the AST. They are analogous to the concrete syntax.

One of the actions can be typing text, however, every string is recognized as it is entered, so there is no tokenizing and the text is being entered into the templates defined in the editor. The resulting changes are displayed to the developer in a newly derived projection of the underlying AST.

The projection uses graphical elements to represent the representation. Although often appearing textual, each of the text elements are references to nodes in the AST.

Developers can only interact with the editor via the rigidly controlled code completion menus or gestures and actions.

The AST is directly built from each interaction she has with the editor. Nodes are creates as instances of the concepts defined in the meta-model. Each node has it's own unique id and points to it's defining concept. It is unambiguous. References are first class and defined by the id rather than resolved by name, as in parser based languages. Disambiguation happens at time of input, as the developer choses from limited legal inputs.

The separation of the abstract and concrete allows the language engineer to implement multiple projections of the same model, using different notations, each node of the AST taking having the design she envisions. The pattern used for projection is similar to MVC, so multiple views of the program can be visible and updateable at the same time.

Graphical modelling tools, for example for UML modelling, are specialized implementations of projectional editing. UML diagrams are not stored as pictures and whose pixels are parsed to create an AST. Instead the model is stored, often with extra information

about visual layout, and the image of the UML is projected to the modeller to edit. Projectional editing generalizes this approach to projecting any notation defined by the language engineer.

What advantages does projectional editing bring?

Projectional editing gives advantages both to the language engineer and the program developers. There follows, in no particular order, a number of advantages of projectional editing synthesised from the literature mapping described in the chapter 3.[TODO: make sure I describe the literature mapping].

Exploratory programming[13–16] As with their progenitors, syntax-directed editors, modern projectional editors help guide a developer unfamiliar with a language. The defined editors with rigid syntax and pre-defined layout mean that only specific cells within the editor can be edited. This template style means the she does not have to worry about significance of spacing or indentation. Minutiae of syntactic adornments, such as statement ending semi-colons or enclosing matched brackets, are also not interfering with her exploration of the language space.

When creating code the developer is only presented with legal options within the current context. As the projection is context aware, relevant actions or options can be suggested and irrelevant ones can be removed. Thus, it is easier for her to explore which options and actions the language allows her to choose. Intelligent code completion does not have to be limited to single nodes. Whole subtrees can be inserted allowing the developer to explore the larger structures of the language.

Correctness-by-construction A projectional editor, by controlling the interaction between the developer and the AST, prevents her from writing syntactically incorrect code. The whole class of syntactical errors are made impossible, with the developer relieved of having to think about special characters and layout. Typing and scoping errors are removed by only allowing validly typed and scoped options for the developer.

The developer is only able to select statements that are legal in the context of the location within the AST. Code does not have to be disambiguated, as this happens at time of entry by the developer. If there are multiple items that share the same presentation in the editor, the developer chooses the relevant item, resolving the ambiguity to what she means rather than what the parser thinks she means.

Rich notation The choice of projection is unconstrained by the restrictions of code that needs to be parsed from a textual source. This freedom opens up diverse otherwise difficult or impossible to parse notations. Examples include tabular, mathematical expressions and symbols, diagrams, trees, images, forms, prose, sub- and superscript. Any visual form or shape that can be mapped to the AST can be used to represent the program in an editor.

With these notations one can better reflect the semantics of the program domain, which should aid comprehension. Mathematics has a rich history of use of notation.

When writing a DSL for the Mathematics domain, the domain experts can interact with it in the centuries old language of their domain.

Of course the projections can also be projections of text. This is often the appropriate projection type if the developer interacting with the language's domain expertise is parser-based languages.

Mixed notation Because no parsing is required, the different forms of rich notation can be combined without the need to create a unifying parser. With all notations working on the same editor infrastructure mathematic symbols can be embedded within textual projections, within tables within graphical representations. As ambiguity is not an issue for the underlying AST, then mixing different notations becomes much easier.

Multiple views With the AST being the stored artefact rather than the notation, projectional editing allows the language engineer to define multiple views on the same model optimised for different tasks. Similar to how software architecture presents different views for different stakeholders interests, the same notational diversity can be achieved with specific editors targeted to experts in the various parts of the domain. A developer can switch between different projections of a node within a larger projection, to find the one that best suits their current task.

Because the architecture of a projectional editor follows the principles of model view controller, it is possible to have multiple simultaneous views of the model. This allows the developer to update a projection that is optimised for writing and immediately see its effect in a projection optimised for understanding.

Language composition Parser-based languages can support some modularization and composition, but a projectional editor allows easy and extensive modular language extension and composition. This is a result of the nodes of an AST being disambiguated at entry rather than through a parser. If two items with the same syntax are available at the same place, then the user will choose the one that they require, and therefore the node has an explicitly chosen meaning.

The composition of independently developed languages does not suffer from the syntactic or keyword clashes they would in two grammar defined languages. Because of the lack of ambiguity, every node referencing the concept that defines it, these languages, when put together, will not have structural or syntactic issues.

Composition can involve extending an existing language or embedding other languages in a host language without modifying the definition of said language. The ease of composition and extensions leads to the advantage of being able to build larger languages out of smaller modules.

IDE functionality Developers in mature languages are used to the functionality of mature IDEs. These functionalities include syntax highlighting, intelligent code completion or suggestion, and static analysis for errors and validation. As projectional languages store the AST rather than the concrete syntax, they require an IDE to edit. Because of this, when a language engineer designs the language she also has to design the IDE.

Because the projection is based on the AST it always knows its context. When the editor already knows the meaning of the node it is representing, then syntax highlighting is simple. From knowing its context, it makes it much simpler to suggest intelligent code completions.

Always having a complete AST makes it much easier to validate scope, typing and other hard to implement code validators.

Language evolution Extending a language with new capabilities and syntax is simple. If the change is syntactic then the language engineer has to update an editor. If there is a semantic change then the language engineer can write a migration in the language, to transform a node of one concept to a different type, and the developer would have to run that migration on their code.

Ancillary data Data can be added to nodes that can augment the AST. This has shown to be useful for documentation, requirements traceability and product line feature dependencies.

This idea emerged as early as in 1970s, but it failed to get adopted widely, mostly due to inconvenient and unnatural way of manipulating code.

The freedom in defining the concrete syntax is not problematic for the computer, but it can cause problems for the language user. Ambiguous syntax is not only a problem for parsers, but a problem that affects humans as well. But there are also downsides to this: syntax ambiguities do not only affect parsers, but humans as well.

This notational flexibility leads to drawbacks. In a ParE, a program can always be typed exactly the way it looks by typing the sequence of characters one by one.

ProjE does not support custom layout (SM.6) – the representation is determined completely by the projection rules.

Historically Projectional editors have had useability challenges and weren't used much in practice.

Traditionally, projectional editors were tedious to use and were hardly adopted in practice.

However, ProjEs have traditionally had two problems. First, for notations that look textual, users expect that the editing behavior resembles classical text editing as much as possible. Second, ProjEs cannot store programs in the concrete syntax – otherwise, this syntax would have to be parsed when programs are loaded into the editor. Instead, programs are stored as a serialized AST, often as XML.

However, the challenge in projectional editing is making the editing experience convenient and productive. Traditionally, projectional editors have had a bad reputation because the user experience in editing programs was unacceptable.

Traditionally, projectional editors were hard to use and were not adopted much in practice.

Projectional editing is often considered a drawback because the editors feel somewhat different and the programs are not stored as text, but as a tree (XML).

Traditionally, PEs also have drawbacks, which is why they have not seen much adoption despite their advantages.

The following two drawbacks are the most important. First, for languages that use a textual syntax, users expect the editor to behave like regular, characteroriented text editors. Since PEs do not work with sequences of characters, this can be a challenge.

The second challenge of PEs is infrastructure integration. PEs do not store programs as text, because this would re-introduce parsing and hence negate the advantages. Instead, the AST is persisted, typically as XML. For use in practice, the integration of these XML files with version control systems must be addressed: diff/merge must be supported using the concrete, projected syntax.

Textual source code can be written in any text editor and simply be shared. However, a projection can still be ambiguous or confusing to the user. Some projections are really bad and should be rejected, which this paper advocates.

Thus, it is crucial that projectional editors do not negatively impact editing efficiency for textual notations.

This is their weak spot: for textual notations, projectional editors behave differently from what developers know from traditional text editors in terms of the granularity and restrictions of code edits and code selections.

The main drawback of projectional editors is their questionable usability. Projectional editing can also be used for a syntax that is textual or semi-graphical (mathematical notations for example). However, since the projection looks like text, users expect interaction patterns and gestures known from "real text" to work (such as cursor movements, inserting/deleting characters, rearranging text, selection).

A projectional editor has to "simulate" these interaction patterns to be usable. However, since the projection looks like text, users expect interaction patterns and gestures known from "real text" to work. For a projectional editor to be useful, it has to "simulate" interaction patterns known from real text.

Because projectional languages by definition need an IDE for editing (it has to do the projection!), language definition and extension always implies IDE definition and extension.

As the examples below will show, projectional editing has a number of intriguing advantages, but there is also a price to pay: this price is the additional effort that goes into defining a nicely usable editor.

This transition requires significant training and it can be a cause of resistance. By improving the editing experience we can reduce this risk.

Traditionally, projectional editors have also had disadvantages relative to editor usability and infrastructure integration; those are discussed in [3]. One characteristic of projectional editors is that the language structure strictly determines the structure of the code that can be written in the editor.

At the same time, projectional language workbenches like MPS [57] and Intentional [47]

Second, Behringer, Palz, and Berger (2017) present Projectional Editing of Product Lines (PEoPL), a DSL that enables multiple projections for variability mechanisms.

Such editors have existed since the 1980s and gained widespread attention with the Intentional Programming paradigm, which used projectional editing at its core.

Projectional editing, also known as structured editing or syntax-directed editing, is not a new idea; early references go back to the 1980s and include the Incremental Programming Environment [32], GANDALF [35], and the Synthesizer Generator [39]. Work on projectional editors continues today: Intentional Programming [44, 18, 45, 14] is its most well-known incarnation. Other contemporary tools [20] are the Whole Platform [9], M'as [3], Onion, and MPS [4]

Projectional Editors from the 1980s. GANDALF [35] and the Incremental Programming Environment (IPE) [32] do not attempt to make editing textual notations efficient; for example, they lack support for linear editing of tree-structured expressions. The Synthesizer Generator [39] avoids the use of projectional editing at the fine-grained expression level, where textual input and parsing is used. While this may improve editing efficiency, it risks the advantages of projectional editing, because language composition at the expression level is limited. Another work that implements and uses a DSL within the Synthesizer Generator [37] concludes: “Program editing will be considerably slower than normal keyboard entry, although actual time spent programming non-trivial programs should be reduced due to reduced error rates.”

The Intentional Domain Workbench (IDW) is the most recent implementation of the Intentional Programming paradigm [44, 18], supporting diverse notations [45, 14]. Since it is a commercial, closed-source project without widespread adoption yet, we cannot easily study it or survey its users.

All contemporary projectional editors are part of language workbenches

An early example of a projectional editor is the Incremental Programming Environment (IPE) [16]. It supports the definition of several notations for a language as well as partial projections, where parts of the AST are not shown. However, IPE did not address editor usability; to enter $2+3$, users first have to enter the $+$ and then fill in the two arguments. Another early example is GANDALF [17]; the report in [20] states that the authors experienced similar usability problems as IPE: “Program editing will be considerably slower than normal keyboard entry, although actual time spent programming non-trivial programs should be reduced due to reduced error rates.” The Intentional Programming project [9, 22] has gained widespread visibility and has popularized projectional editing; the Intentional Domain Workbench (IDW) is the contemporary implementation of the approach. IDW supports diverse notations [7, 23].

Scratch [15] is an environment for learning programming. It uses a projectional editor, but does not focus on textual editing; it relies mostly on nested blocks/boxes. So does GP [18]. Textual notations, and thus grammar cells, are not relevant. Prune [2] is a projectional editor developed at Facebook. The goal is explicitly to not feel like a text editor; the hypothesis is that tree-oriented editing operations are more efficient than those known from text editors. While this is an interesting hypothesis, our considerable experience with using projectional editing in real projects has convinced us that this approach is not

feasible; hence the work described in this paper.

The Synthesizer Generator [21] is a projectional editor which, at the fine-grained expression level, uses textual input and (regular, textual) parsing. While this improves usability, it destroys many of the advantages of projectional editing in the first place, because language composition and the use of non-textual notations at the expression level is limited.

Eco [10] relies on language boxes, explicitly delineated boundaries between different languages used in a single program (e.g., the user could define a box with Ctrl-Space). Each language box may use parsing or projection. This way, textual notations can be edited naturally, solving the usability issues associated with editing text in a projectional editor.

Lamdu [5], a functional, projectional language (no paper)

. Dataflow visual programming languages, such as Blueprints in the Unreal Engine [2], are often domain-specific.

Early syntax-directed source code editors included Interlisp-D (for Lisp's limited syntax) and Emily[1] (for PL/I's rich syntax).

deuce: lightweight structured editing in sketch-n-sketch

Blockly: <https://developers.google.com/blockly>

An early example of a ProjE is the Incremental Programming Environment (IPE) [4]. Another early example is GANDALF [5], which generates a ProjE from a language specification.

The Synthesizer Generator [7] is also a ProjE. However, at the fine-grained expression level, textual input and parsing is used. While this improves usability, it destroys many of the advantages of projectional editing in the first place, because language composition at the expression level is limited. In fact, extension of expressions is particularly important to tightly integrate an embedded language with its host language [8].

The Intentional Programming [2,3] project has gained widespread visibility and has popularized projectional editing; the Intentional Domain Workbench (IDW) is the contemporary implementation of the approach. IDW supports diverse notations [9,10].

Language boxes [11] rely on explicitly delineating the boundaries between different languages used in a single program (e.g., the user could change the box with Ctrl-Space). Each language box may use parsing or projection.

2.2.2 What are Language Workbenches?

2.2.3 What is MPS?

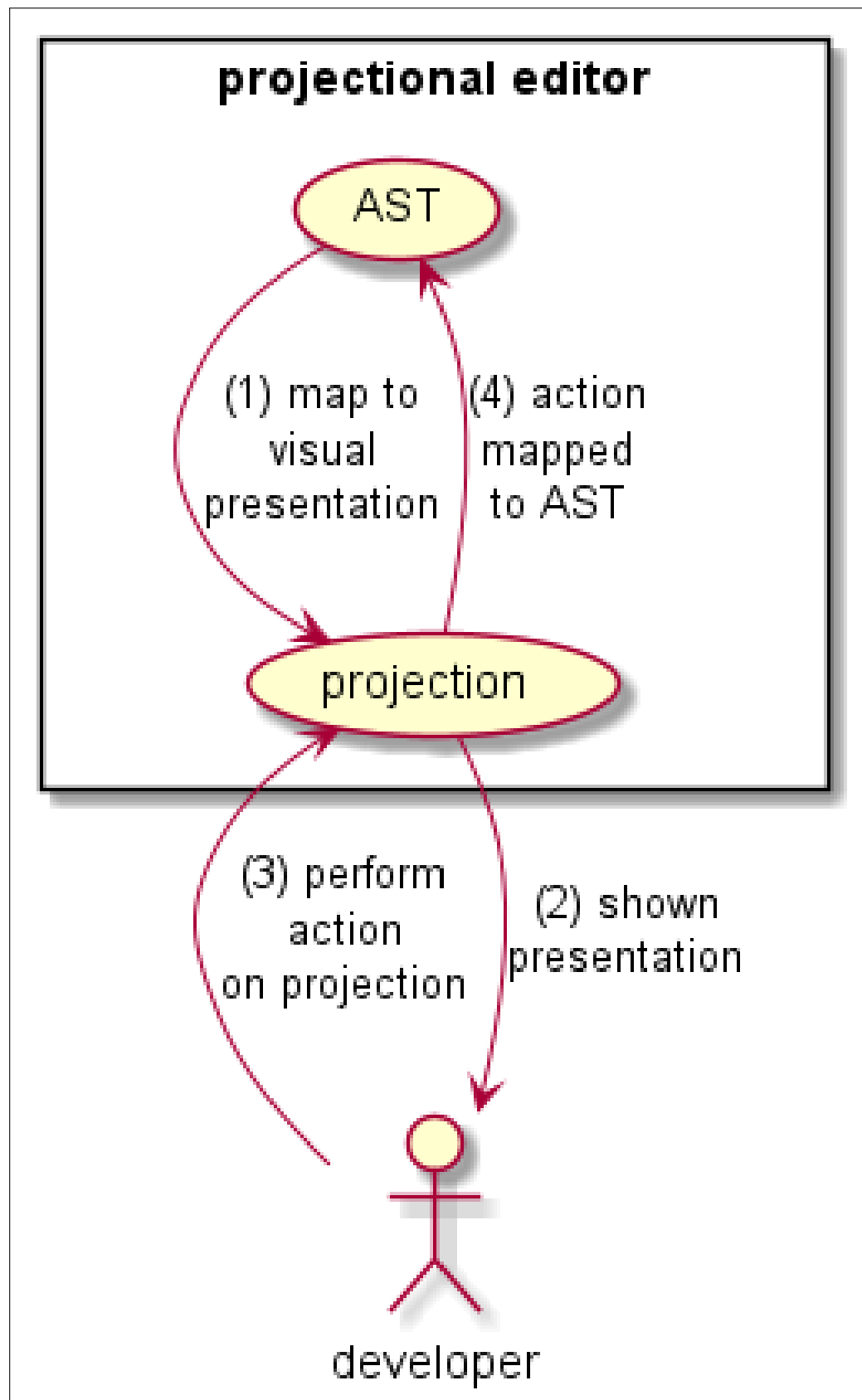


FIGURE 2.4: Projectional editing loop.

Chapter 3

Method

3.0.1 constructing prototypes

We relied on the language Workbench MPS to support our development of a projection of the

Chapter 4

Results

the purpose of abstraction is not to be
vague but to create a new semantic level
in which one can be absolutely precise.

Logico-Tractatus Philosophicus
Edsger W. Dijkstra

Chapter 5

Discussion

5.1 Threats to Validity

5.1.1 Construct Validity

5.1.2 Internal Validity

5.1.3 External Validity

5.1.4 Reliability

5.1.5 Repeatability vs Reproducibility

5.1.6 Method improvement

Chapter 6

Implications to research and practice

6.1 Implications to research

6.2 Future research directions

6.3 Implications to practice

Chapter 7

Conclusion

We have built our projections as an aid to the understanding of Drools rules. This DSL extension includes many different ways to look at and interact with large code bases, as well as presenting options to deal with the complexity of individual rules. This means that they must be [TODO: PROPERTIES THAT AIDS UNDERSTANDING]. Our questionnaires show that we have reached that aim. Since developing our projections we have used them to model complex rules in our host organization.

Two factors lead to our success. First was the flexibility and extensibility of the MPS tool which presented the ability to develop and extend DSLs very efficiently. If we had tried this project without this tooling we would have [TODO: Finish our thoughts] Second [TODO: Finish our thoughts]

In this paper we described our work with first translating the Drools DSL into a projectional language followed by our explorations of projections. We discussed the advantages and disadvantages of the different projections we created and analysed experienced developers reactions to them.

Whilst we are convinced our projections [TODO: finish our thoughts]

Appendix A

Interview Transcripts

Write your appendix content here.

Bibliography

- [1] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information.," *Psychological review*, vol. 63, no. 2, p. 81, 1956.
- [2] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," in *Readings in Artificial Intelligence and Databases*, Elsevier, 1989, pp. 547–559.
- [3] C. J. Date, *What not how: the business rules approach to application development*. Addison-Wesley Professional, 2000.
- [4] M. Fowler, *Should i use a rules engine?* <https://martinfowler.com/bliki/RulesEngine.html>, Accessed: 2021-07-18, 2009.
- [5] P. Jackson, "Introduction to expert systems," 1986.
- [6] E. H. Shortliffe, "Mycin: A rule-based computer program for advising physicians regarding antimicrobial therapy selection.," STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE, Tech. Rep., 1974.
- [7] *CLIPS product page*, <http://www.clipsrules.net/>, Accessed: 2021-07-17.
- [8] *Drools product page*, <https://www.drools.org/>, Accessed: 2021-07-17.
- [9] *BizTalk product page*, <https://docs.microsoft.com/en-gb/biztalk/>, Accessed: 2021-07-17.
- [10] *IBM WebSphere JRules product page*, <https://www.ibm.com/docs/en/iis/11.7?topic=applications-websphere-ilog-jrules>, Accessed: 2021-07-17.
- [11] *OpenRules product page*, <https://openrules.com/>, Accessed: 2021-07-17.
- [12] D. Sottara, P. Mello, and M. Proctor, "A configurable rete-oo engine for reasoning with different types of imperfect information," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 11, pp. 1535–1548, 2010.
- [13] J. Klimeš, "Domain-specific language for learning programming," 2016.
- [14] D. Ratiu, V. Pech, and K. Dummann, "Experiences with teaching mps in industry: Towards bringing domain specific languages closer to practitioners," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, 2017, pp. 83–92.
- [15] M. Völter and E. Visser, "Language extension and composition with language workbenches," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2010, pp. 301–304.
- [16] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *International Conference on Software Language Engineering*, Springer, 2014, pp. 41–61.