applied sciences

MDPI

Article

PyFlies: A Domain-Specific Language for Designing Experiments in Psychology

Igor Dejanović [1,*], Mirjana Dejanović [2,*], Jovana Vidaković [3] and Siniša Nikolić [1]

[1] Faculty of Technical Sciences, University of Novi Sad, 21000 Novi Sad, Serbia; sinisa_nikolic@uns.ac.rs
[2] Faculty of Medicine, University of Priština-Kosovska Mitrovica, 38220 Kosovska Mitrovica, Serbia
[3] Faculty of Sciences, University of Novi Sad, 21000 Novi Sad, Serbia; jovana@uns.ac.rs
* Correspondence: igord@uns.ac.rs (I.D.); mirjana@uns.ac.rs (M.D.); Tel.: +381-21-485-4562 (I.D.)

Abstract: The majority of studies in psychology are nowadays performed using computers. In the past, access to good quality software was limited, but in the last two decades things have changed and today we have an array of good and easily accessible open-source software to choose from. However, experiment builders are either GUI-centric or based on general-purpose programming languages which require programming skills. In this paper, we investigate an approach based on domain-specific languages which enables a text-based experiment development using domain-specific concepts, enabling practitioners with limited or no programming skills to develop psychology tests. To investigate our approach, we created PyFlies, a domain-specific language for designing experiments in psychology, which we present in this paper. The language is tailored for the domain of psychological studies. The aim is to capture the essence of the experiment design in a concise and highly readable textual form. The editor for the language is built as an extension for Visual Studio Code, one of the most popular programming editors today. From the experiment description, various targets can be automatically produced. In this version, we provide a code generator for the PsychoPy library while generators for other target platforms are planned. We discuss the language, its concepts, syntax, some current limitations, and development directions. We investigate the language using a case study of the implementation of the Eriksen flanker task.

Keywords: domain-specific language; textual notation; source code generator; code editor; psychology tests builder

Citation: Dejanović, I.; Dejanović, M.; Vidaković, J.; Nikolić, S. PyFlies: A Domain-Specific Language for Designing Experiments in Psychology. *Appl. Sci.* **2021**, *11*, 7823. https://doi.org/10.3390/app11177823

Academic Editor: Paolino Di Felice

Received: 6 July 2021
Accepted: 20 August 2021
Published: 25 August 2021

1. Introduction

It is hard to imagine work in a modern psychology laboratory without the use of personal computers. These ubiquitous devices are becoming more and more capable each year, being able to present experimental stimuli as well as to record a participant's response with great accuracy. In the past, the development of psychological experiments required a great effort and a lot of low-level coding. Fortunately, things have changed in the last two decades and nowadays we have an array of freely available open-source software tools and libraries that make the process of developing psychological tests much easier.

Based on the experiment specification approach, we can divide the existing software into two categories: (1) GUI-based and (2) text/code-based software.

GUI-based experiment builders use a click/drag&drop interface to construct an experiment (e.g., OpenSesame [1], E-Prime [2], PsychoPy Builder [3], Lab.js [4], Gorilla [5]). This style is a good fit for practitioners without a programming background. Those builders are easy to learn by navigating the interface using a mouse and trying options, i.e., they are suitable for trial and error learning. They often offer a good overview of the experiment structure, although for the details the users usually have to drill down into the dialogs or other GUI elements.

Text-based builders use textual languages as a way to specify the experiment (e.g., PsychoPy [3], Expyriment [6], jsPsych [7], PEBL [8]). While they are initially not that easy

to start compared with GUI builders, they offer some benefits. First, we can leverage all the development in text editing in recent decades (syntax highlighting, code completion, tooltips, quick fixes), which usually can be configured in contemporary editors without much effort. Second, standard text editing idioms can be used such as copy–paste, code commenting, etc. Third, the representation is the same as the storage format, which enables the usage of any text editor to read and edit the experiment. Fourth, plain text is perceived as a future-proof format. If the text-based builder ceases to be developed, we can still open and read experiments in the future using a plain text editor. Additionally, last but not least is a collaboration which is the key in today's development. Text-based syntaxes facilitate the usage of standard version control tools such as git [9] and mercurial [10], and services such as GitHub and GitLab to enable distributed collaboration and tracking of the history of changes. Indeed, distributed version control systems have started a revolution in the global collaboration in software development. The GitHub Octoverse report from 2020 (https://octoverse.github.com/, accessed on 1 July 2021) shows that GitHub has over 56 million developers who added over 1.9 billion contributions in the year 2020. The report shows that the numbers are increasing rapidly year after year. It is hard to achieve this level of collaboration with GUI-based experiment builders as their storage formats differ from the display format. While the user interacts with the tool using some form of graphical syntax, the experiment is stored in a format that the user is not familiar with (e.g., XML, JSON, or binary), which requires the development of custom tools for version control. This hampers the merging of concurrent changes and the investigation of the history of changes.

Textual languages can be divided into two categories: (1) General-Purpose Programming languages (GPLs) and (2) Domain-Specific Languages (DSLs). GPLs are programming languages that can be used to build general computer software (e.g., Python, JavaScript, C). The benefits of experiment builders based on GPLs are: (a) they have access to the full host language that offers unprecedented power and flexibility; (b) a user may use any additional library available for the language; (c) existing tools, debuggers, editors, etc., for the host language can be used in the context of experiment builder library as well. However, their usage requires programming skills. Additionally, the essence of the experiment design is not apparent from the implementation code. Conversely, DSLs are expressive languages tailored for a specific domain [11]. They use concepts from the target domain and can only be useful in that domain, i.e., they trade generality for expressiveness in a limited domain [12]. It is a common belief that DSL programs are easier to understand and maintain than GPL programs and that using DSLs is more efficient and effective. This has been empirically validated in several recent studies [13–15].

One of the recurring ideas in the history of DSLs is the involvement of non-programmers, or lay programmers [16]—domain experts who are not professional programmers but program in DSLs—directly in the development of a solution. This idea is also known as end-user development [17]. Although this level of involvement of domain experts is not always possible, it is important for the DSL to provide a syntax that can be read and comprehended by domain experts even if they do not type the program directly. A DSL that can be read by domain experts can serve not only as a design and implementation tool but also as a requirements elicitation medium.

DSLs have been successfully used in various domains. However, we find that in the domain of psychological test specification, while there exist numerous GUI-based builders, text-based builders are mostly based on general-purpose languages.

We believe that DSLs, if properly designed, have the potential to bridge the gap between GUI builders and GPL-based builders, enabling non-programmers to specify experiments while still using the benefits of text-based distributed collaboration and version control.

In the rest of the paper, we present PyFlies, an open-source DSL for implementing psychological experiments. The language design principles are oriented towards readability and ease of use by employing abstraction and first-class domain concepts. We have used

PyFlies so far in educational settings. It is used to teach design of psychology experiments to students of medicine at the University of Pristina-Kosovska Mitrovica and students of psychology at the University of Novi Sad. It is also used to teach the design and implementation of domain-specific languages to students of computer science and software engineering at the University of Novi Sad. PyFlies is available as open-source software under the terms of the GPL 3.0 license. It is hosted at GitHub (https://github.com/pyflies, accessed on 1 July 2021).

The paper is organized as follows: Section 2 gives some theoretical background of the field of DSL. The PyFlies architecture and design principles are described in Section 3. In Section 4, PyFlies language abstract and concrete syntaxes are described. Code generators are described in Section 5. The results given in the form of a case study on the development of a real psychological experiment are given in Section 6. The discussion is given in Section 7. Section 8 presents the related work from the fields of DSL and psychology test builders. Section 9 concludes the paper.

2. Theoretical Background

This research is an application of the domain-specific language approach to the development of tests in psychology. Thus, in this section, we give a theoretical background of the DSL approach to lay the groundwork for our work.

Domain-specific languages (DSLs) are small, usually declarative languages that allow developers to write code at the appropriate level of abstraction using concepts from the domain of their expertise [18]. They provide an effective interface for the domain experts to specify solutions. DSLs can be regarded as a specialization of a more general approach called Model-Driven Engineering (MDE) [19,20]. DSLs put emphasis on domain specificity. DSLs are constrained to the given domain and can be used only to specify solutions in the given domain. This restriction makes DSLs more expressive and concise [12]. Furthermore, the DSL approach fosters the building of a community of domain experts who *speak the same language* [21].

In contrast, general-purpose languages can be used to create arbitrary software solutions. Thus, they use concepts of programming paradigms they conform to such as functions, classes, objects, parameters, methods, threads, calls, loops, conditions, etc. During the design and implementation of the solution, the concepts of the domain must be mapped to computing concepts of the GPL. This mapping is a manual process and the source of difficulties during the development and maintenance of the software [22]. DSLs make the mapping problem non-existent as they offer one-to-one mapping by directly operating in terms of domain concepts.

The artifact that conforms to the given DSL is called *program* or *model*. While some researchers see a difference between the two, we take the stance of the other group that treats both programs and models the same. To emphasize this equality, Kleppe suggested the term "mogram" in [23]. However, the suggested term has not seen a wider adoption so far. Thus, in this paper, we use the terms "program", "model", and "specification" as synonyms.

In the past, DSLs have been known as "little languages" [24], although depending on the domain, these languages do not have to be little in terms of the number of language constructs. However, as the focus of the language expands there is a danger that the language will lose the conciseness and expressiveness of DSL. Indeed, Fortran and Cobol started as DSLs in the area of scientific computing and business computing, respectively. However, they gradually evolved more generality over time [18,25]. We should point out that the domain specificity is not an absolute but a relative measure. Domain specificity is a matter of degree [12]. Additionally, when discussing whether a language is or is not a DSL, we must know what domain we are talking about. For example, JavaScript is regarded as a GPL, but if our domain is "web applications development", then the language could be considered as a DSL. Of course, this is just a conceived example of a too broad domain to

be usable. In the context of this paper, we assume the domain of creating experiments in psychology.

Solutions based on DSLs are easier to maintain [26], validate and verify [27]. In general, whereas in GPLs AVOPT (Domain-specific Analysis, Verification, Optimization, Parallelization, and Transformation) techniques [12] are hard to achieve, in DSLs they are straightforward. Despite their proven benefits, DSLs have been seldom constructed in the past by smaller communities of developers. The reason often cited in the literature [16] is the cost of building and maintaining the language and the supporting tool-chain (parsers, editors, validators, generators, etc.). However, in the last two decades, a new breed of powerful language engineering tools has emerged which reduced this cost significantly and led to their uptake in DSLs applications. These tools have been developed as integrated environments for language engineering, fostering the fast development of languages and supporting services (editors, debuggers, visualizers, etc.). These tools are usually referred to as *language workbenches*, a term coined by Martin Fowler [28]. Well-known text-based representatives of this kind are xText [29], Spoofax [30], and Rascal [31].

DSLs can be classified by the target audience on *technical* or *horizontal* DSLs and *application domain* or *vertical* DSLs ([32] p. 26). Technical DSLs are used by software developers to facilitate the development of a particular aspect or a particular type of software. For example, well-known technical DSLs are SQL (Structured Query Language)—for querying relational databases, or CSS (Cascading Style Sheets, https://www.w3.org/Style/CSS/Overview.en.html, accessed on 1 July 2021)—for defining the styling of the content on the web. On the other hand, application domain DSLs are intended to be used by non-programmers (DSLs for law, healthcare, finance, etc.). PyFlies belongs to the application domain category.

In addition to the approach based on text parsing, there is an approach based on so-called "projectional editing" where the user interacts directly with an abstract representation of the model through the "projection surface" [32]. The advantage of this approach is that arbitrary types of concrete syntax can be supported (e.g., graphical, textual, tabular). Different concrete syntaxes can even be mixed in the same representation. Moreover, the model validation happens during editing so the interaction with the user is richer. The downside is that the tools that support this approach are more complex, and similarly to all GUI-based builders, the storage format differs from the presentation form so standard text-based version control tools and plain-text editors cannot be used. Probably the best freely available tool in this category is *Meta-Programming System* (MPS, https://www.jetbrains.com/opensource/mps/, accessed on 1 July 2021) [32].

The work presented in this paper builds on top of our previous research in the development of tools for building DSLs in the Python programming language [33,34]. These tools provide a fast turnaround and easy modification and evolution of the language's abstract and concrete syntax.

Each DSL consists of three ingredients: abstract syntax, one or more concrete syntaxes, and semantics [32]. The abstract syntax defines the language structure, its concepts, and their relationships (Section 4). The concrete syntax specifies how the language appears to the user and how the user can interact with the models. We can think of concrete syntaxes as interfaces to the users. Concrete syntaxes come in different forms, e.g., textual, graphical, table-based, dialog-based ones. In this paper, we use the textual syntax (Section 4). The semantics of the language defines the meaning of programs written in the given DSL. There exist different approaches to defining semantics [35]. In this paper, we use a pragmatic approach where we map our programs to the target platform whose semantics is already defined. This is achieved using source code generators (Section 5).

3. PyFlies Architecture

This section describes the architecture of PyFlies. To better understand the reasoning behind architectural decisions, in the first subsection, we present the design principles which were followed during the development of the language and the tool.

3.1. PyFlies Design Principles

The design of the PyFlies language was guided by the following principles:

1. **Readability**: Researchers share experiments with their peers. Furthermore, it is a common belief that developers spend more time reading the code than writing it. That means the experiment descriptions should be as easy to read and understand as possible. One of the principles of Python Zen is "readability counts", which is arguably one of the design principles that had the most impact on its adoption.
2. **Abstraction**: PyFlies tries to hide as much of the implementation details as possible. Thus, in PyFlies experiment descriptions, most often, we use abstract terms. For example, instead of using explicit screen coordinates, we often use terms `left`, `right`, `up`, `down`, and let the compiler complete the mapping. These mappings, if needed, can be overridden. In different contexts, these abstract terms may mean different things. For example, a symbol `left` might be a position on the screen, a direction of the pointing arrow image stimulus, or a left arrow key on the keyboard. The code generator is capable of carrying out this context-aware mapping while the experimenter may still use the same term in all contexts. Another abstraction used in PyFlies is abstracting how exactly to carry out various steps in the experiment. The experimenter is more focused on providing information on *what* needs to be achieved than *how* to do it.
3. **Domain concepts**: PyFlies language constructs are domain-specific. For example, instead of having concepts of general-purpose programming languages such as *functions*, *classes*, *methods*, etc., in PyFlies we use *tests*, *screens*, *components*, *timings*, *condition tables*, etc. In our experience, this reduces mental effort in designing and understanding the experiment.
4. **Modular architecture and extensibility**: PyFlies follows a modular approach where target platform code generators can be developed as separate Python projects (Section 3). Moreover, components used in the experiment description are specified using a component description DSL, which will make it possible to open this language to experimenters and enable custom component specification in the future (Section 4.4).
5. **Best practice**: The code generator should be written by developers that are knowledgeable about both PyFlies and the target platform. Thus, the current best practices (at least to the best of generator authors' knowledge) for the target platform are automatically provided for all PyFlies users. Since generators can be improved constantly, all PyFlies experiments undergo improvements for free.
6. **Focus on the experiment structure**: PyFlies specifications are strongly focused on the experiment structure. Each experiment consists of *screens* and *tests* which are executed during the *flow* of the experiment. That is something often found in GUI builders but lacking in textual languages. Experiment [6], implemented as a Python library, is a notable text-based solution that employs this idea.

3.2. Modular Architecture

Figure 1 shows the architecture of PyFlies and its modular design. PyFlies follows the standard compiler architecture that consists of:

- *front-end* (colored green in Figure 1)—parses input files, performs syntax and semantic checks and constructs the intermediate representation;
- *back-end* (colored red in Figure 1)—consists of a set of code generators for different target platforms.

3.3. Front-End

The core PyFlies project implements language *front-end* and the infrastructure for discovering and running generators. We use textX [33], a Python tool for the development of domain-specific languages based on Parsing-Expression Grammars formalism [36] and built on top of the Arpeggio parser [34]. TextX parses the input PyFlies files, performs syntax checks based on the provided grammar, and produces intermediate in-memory

graph representation, which is further checked for semantics conformance. This intermediate representation is further transformed to the target run-time platform by the selected back-end (code generator).

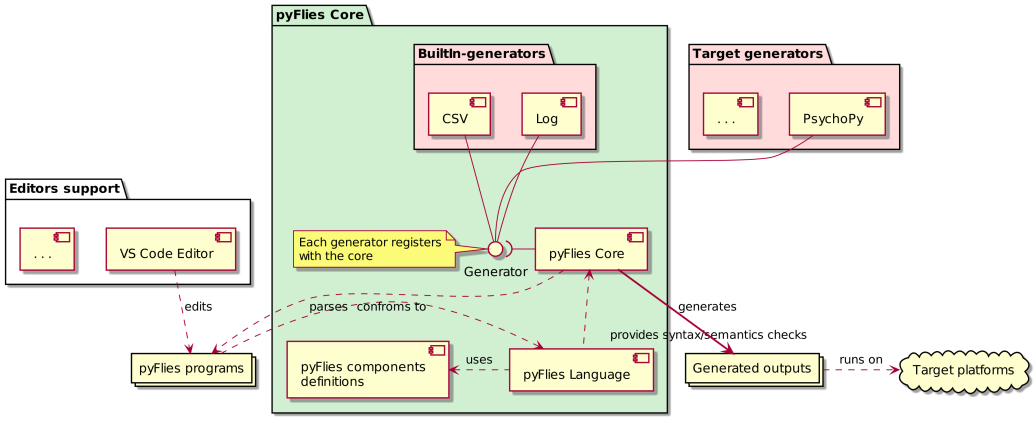

Figure 1. The PyFlies architecture.

3.4. Back-End

PyFlies modular design enables the development of generators as separate projects. By installing Python generator projects, using the standard Python `pip` package manager, we make them available to the PyFlies core. We directly utilize textX's generators and languages registration and discovery support (http://textx.github.io/textX/latest/registration/, accessed on 1 July 2021). This enables listing and using generators from the command line interface (CLI) by the `textx` command, but also discovering and using generators programmatically via textX registration API.

The PyFlies core provides two generators out-of-the-box: CSV (*Comma Separated Values*) and Log. These two generators are generic enough to be supported by the core itself. CSV generator produces a `.csv` file with the data from the expanded test table (Section 4.3), whereas the Log generator produces log files with detailed information about the flow of the experiment. Although the PyFlies core project implements these two generators, they are still registered, discovered, and run the same way as all other externally implemented generators. We discuss generators in more detail in Section 5.

4. Language Abstract and Concrete Syntaxes

In this section, we give a brief overview of the core language constructs. A detailed description of the language is available in the PyFlies documentation (https://pyflies.github.io/pyflies/, accessed on 1 July 2021).

4.1. Language Abstract Syntax

A simplified abstract syntax of the language is given in Figure 2. An abstract syntax of a language is also known as *the meta-model* in the modeling communities. The full meta-model of the PyFlies language is available in the project GitHub repository, https://github.com/pyflies/pyflies/tree/main/docs/metamodel, accessed on 1 July 2021. The diagram shows that each `PyFliesModel` contains a `Flow` definition, one or more routines represented by the `RoutineType` concept, and zero or more `Target` definitions. Routine can either be a test, represented by the `TestType` concept, or a screen, represented by the `ScreenType` concept. Each test has a `ConditionTable` and zero or more instances of `Component`.

A more detailed description of each relevant concept together with its concrete syntax is given in the following subsections.

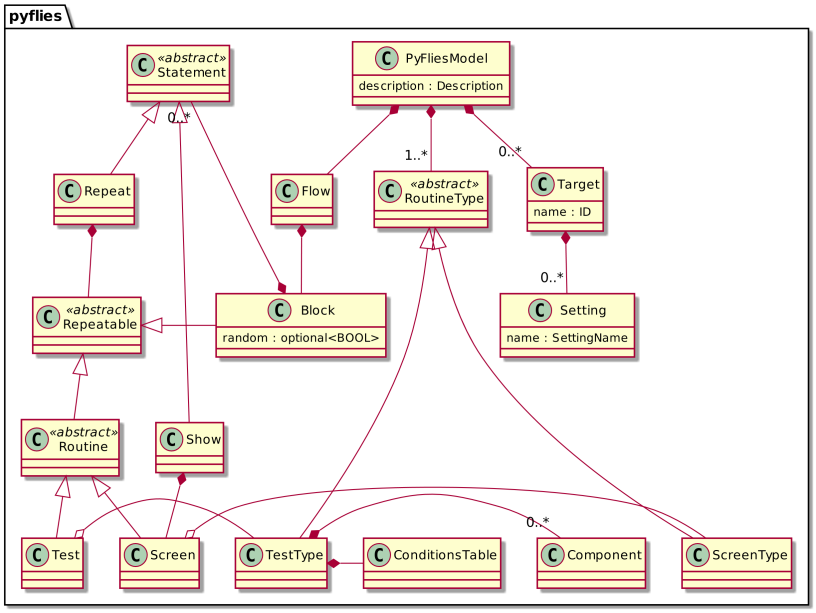

Figure 2. A simplified PyFlies abstract syntax.

4.2. Tests

Test is the core concept of the PyFlies language. Each test consists of a condition table and trials' components specification.

The definition of `Test` concrete syntax in textX grammar language is given in Listing 1. `TestType` grammar rule specifies test definition that starts with keyword `test` followed by attribute `name` matched by the textX built-in rule `ID`. It is further followed by the test body that starts with the literal string match `"{"` (line 2). The body of the test starts with condition table specification which is matched by the `ConditionsTable` grammar rule (Section 4.3). After the condition table, we have optional variable assignments matched by the rule `VariableAssignment`. At the end, we have component conditions specifications represented by textX attribute `component_cond` and matched by the rule `ComponentsCondition`. The closing curly brace ends the test body.

Listing 1. textX grammar of the `Test` concept.

```
1 TestType:
2   "test" name=ID "{"
3       table_spec=ConditionsTable
4       vars*=VariableAssignment
5       components_cond*=ComponentsCondition
6
7   "}"
8 ;
```

Listing 2 shows an example of a test definition (the used example test is a parity judgment task [37]; a complete test is given in the `examples` folder of the PyFlies project repository.). Test definition starts at line 4 with the keyword `test` and the name of the test `Parity`. The first part of the test definition is a condition table defined at lines 6–8, which describes trial variables (`number` and `parity`) and their values for each trial (line 8) in a compact form (see Section 4.3 for details).

Listing 2. An example of a test definition.

```
1 numbers = 1..9
2 parities = [odd, even]
3
4 test Parity {
5
6     | number        | parity    |
```

```
    | ------------ | -------- |
    | numbers loop | parities |

    fix -> cross() for 700
    exec -> text(content number)
            keyboard(valid parities, correct parity)

    correct -> sound(freq 500) for 300
    error -> sound(freq 1000) for 300
}
```

The second part of the test definition specifies trial phases with component specifications.

PyFlies divides each trial into three phases: *fix*, *exec*, and *error/correct*. The fix phase is a fixture phase during which a fixture stimulus is shown to the subject. After the *fix* phase, we have the *exec* phase that is the main phase of the trial and the only mandatory phase (all others are optional). Depending on the subject's response, the third phase can be either *correct*, if the response was correct, or *error*, if the response was incorrect. We provide the correct response in the definition of the input components (e.g., `correct` property of the `keyboard` component is set to a symbol or a list of symbols describing the correct response).

We provide each component specification in the following form:

```
<boolean expression>
        -> <component specifications with timings>
```

The Boolean expression on the left side, when evaluated as true, enables the execution of components specified on the right side. There can be more components defined for each Boolean expression.

During the execution of each phase Boolean variables `fix`, `exec`, `error`, and `correct` will have a corresponding value. For example, `exec` will be `true` during the *exec* phase and `false` in all other phases. That enables creating Boolean expressions on the left side of each trial definition expression that matches a particular phase. For example:

```
fix -> cross() for 700
```

Will match the fixation phase and show a `cross` component for the duration of `700 ms` in this phase.

The optional duration is specified after the keyword `for`. It is given as a PyFlies expression that evaluates to an integer interpreted as a duration in milliseconds. If the duration is not given the component runs until the end of the trial phase.

Most of the time a Boolean expression on the left side is just a phase variable but can be an arbitrary Boolean expression. For example:

```
fix and parity == odd
    -> circle(color green) for 700
```

Will show a green circle in the *fix* phase if `parity` is `odd`.

4.3. Conditions Tables

Condition tables are used to describe trial variable values in a compact form. Each column of a condition table specifies a variable available in each test trial.

The heading row of the table defines trial variables, whereas the remaining rows specify variables' values and thus define the state of each trial. We could write a row for each trial with the exact values, thus defining the table in expanded form. Instead, we usually specify values using expressions that are evaluated during compilation yielding the fully expanded condition table. That brings more flexibility in extending the table, both column-wise and row-wise.

Listing 3 gives the definition of a condition table concrete syntax in textX. `ConditionsTable` uses a `noskipws` rule modifier as we want to control whitespaces ourselves in this rule. The condition table can be specified in either Org Mode [38] or markdown table format (https://www.w3schools.io/file/markdown-table/, accessed on 1 July 2021). At

the beginning of the `ConditionsTable` rule (line 1), we have a match that consumes all whitespaces and comments (line 2) until the first pipe char (`|`) that starts the table. Line 3 defines the syntax of the table header with variable names. It starts with the pipe char, after which we have one or more (`+=`) variable name matched by the rule `TableVarName`. This variable assignment uses separator `'|'` textX repetition modifier during the matching (part `['|']` after `TableVarName`). Rules `WSWithNL` at line 8 and `WSNoNL` at line 9 match whitespaces with newline and whitespaces without a newline, respectively. The expression at line 4 consumes the header separator. The header separator is not preserved (i.e., no textX assignment is used). After the header, one or more condition specification is consumed (line 5) matched by the rule `Condition` (not given in the listing for brevity).

Listing 3. textX grammar of the `ConditionsTable` concept.

```
1 ConditionsTable[noskipws]:
2     (/\s*/ Comment? /\s*/)*
3     '|' variables+=TableVarName['|'] WSNoNL '|' WSWithNL
4     WSNoNL /\|\s?(:?-*:?\s?(\||\+)\s?)*:?-+:?\s?\|/ WSWithNL
5     cond_specs+=Condition
6 ;
7
8 WSWithNL: /[ \t]*\r?\n/;
9 WSNoNL: /[ \t]*/;
```

Listing 4 shows two tables with three variables: `color`, `direction` and `congruent`. The first table is an expression-based form of a condition table. We use `loop` expressions in the first two columns and a Boolean expression in the third column. `loop` expressions will produce a row for each value from the collection it is called upon (in this case, of the `list` type). `loop` expressions are evaluated from left to right.

Listing 4. A condition table with its expanded version.

```
| color                  | direction          | congruent      |
| ---------------------- | ------------------ | -------------- |
| [red, green, blue] loop | [left, right] loop | color == green |

// expands to
| color | direction | congruent |
| ----- | --------- | --------- |
| red   | left      | false     |
| red   | right     | false     |
| green | left      | true      |
| green | right     | true      |
| blue  | left      | false     |
| blue  | right     | false     |
```

The second table is an expanded form of the first table. We see that we have each color from the list of colors iterated in the first column. For each color in the first column, a list of directions from the second column is iterated. And lastly, the third column's Boolean expression is evaluated in the context of each row.

Both styles of condition table are valid. We can use either, but the expression based form is usually shorter and more flexible.

PyFlies condition tables use markdown table syntax, which enables convenient editing and good readability while still being pure text. However, for the best user experience, an editor with markdown tables support should be used. PyFlies extension for Visual Studio Code (VS Code) editor offers this feature.

Besides, test definitions tables are also used in `repeat with` statements in the `flow` block (Section 4.6) where they represent variables' values for each repetition of the loop.

4.4. Components

The interaction with the experiment participants during the test is carried out through components. There are currently 11 components defined by the PyFlies. Additional components are easily specified.

We describe PyFlies components using a built-in DSL for components description. The description consists of a component name, documentation, and a set of properties with names, types, default values, and their documentation. A component can inherit other components.

A textX grammar for components definition is given in Listing 5. The optional `abstract` keyword (line 2) specifies components that can only be inherited but not referenced in tests. A component can inherit multiple other components. Inherited components are specified after the keyword `extends` (line 3). The component header is followed by its description after which the body is defined. The body of the component consists of zero or more parameters defined by the `ParamType` rule (line 6). The `ParamType` rule defines component parameters (line 10). It starts with the keyword `param` followed by the `name` attribute that is matched by the `ID` rule. The parameter type is specified after the colon and it can be a simple type or a list of types inside square brackets (line 12). Multiple types are used in a situation when the parameter may be one of several types (e.g., a `symbol` or a `point`, see Listing 6). The default parameter value is given after the char =. At the end of the definition, an optional description may be provided.

Listing 5. A textX grammar for components definition.

```
1  ComponentType:
2      abstract?='abstract' 'component' name=ID
3          ('extends' extends+=[ComponentType][',']) ?
4      (description=Description)?
5      '{'
6          param_types*=ParamType
7      '}'
8  ;
9
10 ParamType:
11     'param' name=ID ':'
12         (types=Type | '[' types+=Type[','] ']') '=' default =
               Expression
13     (description=Description)?
14 ;
```

Listing 6 shows a definition of the `rectangle` component. This component inherits a `visual` component that describes common properties used by all visual components and adds property `size` which can be of either `symbol` or `point` type. The default value is point `(20, 20)`. This definition provides information for semantic checks of components' usage in test specifications.

Listing 6. PyFlies `rectangle` component defined in the internal DSL.

```
component rectangle extends visual
"""
Visual stimuli in the form of a rectangle.
"""
{
    param size:[symbol, point] = (20, 20)
    """
    Override ‘size‘ to be of point type representing width and height.
    0 for height means 'keep aspect ratio'.
    """
}
```

We use the component description language internally in the core PyFlies project, but we plan to open this language to end-users, which will enable them to specify custom components. However, specifying and using new components will require users to extend the target generator they use as well.

Note that component descriptions do not specify their full semantics, e.g., they do not specify what components do and how they operate during the experiment run. That is entirely left to the code generator to decide. For example, there is no fundamental difference in the definition of visual components, such as `circle` or `rectangle`, and input

components, such as `mouse` and `keyboard`. They are all defined just by their names, inherited components, and a set of properties. A consequence of this design decision is that we can introduce a new class of components in the future, either by the core project or the language users, without the need to change the core.

4.5. Screens

During the experiment, we often need to show messages to the subject, e.g., by describing what they are supposed to do or providing feedback about their performance. For this purpose, PyFlies provides `screen` language construct (see Listing 7 for an example). The screen definition is a simple construct that starts with the `screen` keyword, followed by the screen name and a block of text within curly braces. The text in the body of the screen statement is often plain text but may have dynamic parts. This dynamic expansion is implemented by passing the body text through the Jinja template engine [39], which is one of the most popular Python template engines available. In general, a template engine enables template-based code generation by mixing static text, which is left unchanged, and the text dynamically created by template language expressions [40]. In the example screen given in Listing 7, a dynamic part of the text is `{{real_or_practice}}` which renders the `real_or_practice` screen parameter, enabling the same screen to be used both for practice and real trials block.

Listing 7. An example `Screen` definition.

```
screen Intro {
    Parity classification
    ---------------------

    This is a {{real_or_practice}} block.

    You will be presented with a digit.
    Press
    the LEFT key for odd and
    the RIGHT key for even digit.

    Press ENTER key for the start.
}
```

4.6. Flow

The `flow` part of the experiment specification defines the sequence of test executions and screen displays. The `flow` construct starts with the keyword `flow` followed by the body of the statement enclosed in curly braces (Listing 17).

The `flow` statement body contains a sequence of statements for executing tests, showing screens, or repeating a test or a statements block.

To show a screen you use `show` statement of the form:

```
show Intro for 10000
```

The previous statement displays the `Intro` screen for 10 seconds. The `for` part is optional, and if not provided, the screen will be displayed until a keypress.

To execute a test, you use the `execute` statement of the form:

```
execute Parity(practice true,
               random true, some_param 42)
```

Note the parameters given in the parentheses. Parameters can be specified in both `execute` and `show` statements and, if provided, can be referenced inside tests and screens. Parameters can be named arbitrarily but `practice` and `random` names have special treatment — the former is used to denote if data should be collected, while the latter is used to designate if the trials should be randomized.

To repeat a test or a block of statements multiple times, we use the `repeat` statement. This statement can apply to a test. For example:

```
repeat 5 times Parity(practice true, random true)
```

Or it can apply to a block of statements and can nest at an arbitrary depth. For example:

```
repeat 3 times {
    screen Instructions for 10000
    repeat 2 times {
        screen InnerBlockInstructions
        execute Posner
    }
}
```

Finally, you can repeat the block over a condition table using `repeat...with` statement (Listing 8). This statement has a condition table at the end, and it repeats the given block of statements for each row of the expanded table.

Listing 8. A repeating block of statements with a condition table.

```
repeat {
    show instruction

    // 3 same blocks
    repeat 3 times {
        execute showImages
        show break for 1000
    }
} with
| image_type       | order |
| ---------------- | ----- |
| image_types loop | 1..2  |
```

In Listing 8, the outer `repeat` loop will repeat as many times as there are elements in the `image_types` list used for the `loop` in the condition table (Section 4.3).

4.7. Targets

If a target code generator needs configuration, we can provide it in the `target` section. This section starts with a keyword `target` followed by the name of the target platform and the body enclosed in the curly braces. Within the target body, we specify key-value pairs where the key is a configuration variable name.

In Listing 9, we see a configuration, which configures `background`, a configuration parameter used to choose the background color, and a mapping between parities used in the experiment and the keys on the keyboard.

Listing 9. An example configuration for the PsychoPy target generator.

```
target PsychoPy {
    background = grey
    odd = left
    even = right
}
```

5. Code Generators

The PyFlies back-end consists of a set of target platform's source code generators, which are developed as separate projects. The PyFlies core discovers all registered generators at run-time using textX registration mechanism [3,4] based on Python `setuptools` (https://setuptools.readthedocs.io/, accessed on 1 July 2021) *entry points* concept. The entry point used for textX generators is `textx_generators`. Listing 10 shows the declarative registration of the PsychoPy generator using `setup.cfg` file. The generator is described in more detail in Section 5.2.

Listing 10. Registering the PsychoPy generator in `setup.cfg` using the `textx_generator` entry point.

```
[options.entry_points]
textx_generators =
    pfpsychopy = pfpsychopy:pyflies_generate_psychopy
```

Each textX generator is a Python callable (Listing 11) registered using `@generator` Python decorator that is parameterized by the textX language name and the target platform name (line 1). The generator function accepts textX metamodel, the model created by textX from the input program file, output path, `overwrite` and `debug` flags, and the custom generator parameters (for more information, please see textX docs on generators — http://textx.github.io/textX/stable/registration/#textx-generators, accessed on 1 July 2021) (lines 2–3). The generator function is free to generate the target in whatever way it chooses. It is entirely left to the generator's authors. However, textX provides optional integration with the Jinja template engine (Jinja integration is supported through the `textX-Jinja` project—see http://textx.github.io/textX/stable/jinja/, accessed on 1 July 2021) that makes creating generators easier. The supported Jinja integration is used by all PyFlies generators described in this paper.

Listing 11. A part of the generator function for the PsychoPy target.

```
1  @generator('pyflies', 'psychopy')
2  def pyflies_generate_psychopy(metamodel, model, output_path,
3                                overwrite, debug, **custom_args):
4      "Generator for generating PsychoPy code from
5  ␣␣␣␣␣pyFlies descriptions."
6
7      ... augmenting input model, preparing Jinja template file
8      ... and filters, preparing 'settings' dict,
9      ... calculating output path, etc.
10
11     now = datetime.datetime.now()
12     now = now.strftime('%Y-%m-%d %H:%M:%S')
13     config = {'m': model, 's': settings, 'now': now}
14
15     # call the Jinja based generator
16     textx_jinja_generator(template_file, output_file, config,
17                           overwrite, filters)
```

After the generator is registered, it is discovered by textX and can be used both programmatically through the textX registration API or by the `textx` CLI command. For example, listing all registered generators can be carried out by the `textx list-generators` command, as shown in Listing 12.

Listing 12. Listing all registered textX generators.

```
(env)$ textx list-generators
any -> dot            textX[2.3.0]    Generating dot visualizations
                                      from arbitrary models
textX -> dot          textX[2.3.0]    Generating dot visualizations
                                      from textX grammars
textX -> PlantUML     textX[2.3.0]    Generating PlantUML
                                      visualizations from textX grammars
pyflies -> csv        pyflies[0.4.2]  Generator for CSV files from
                                      PyFlies tables.
pyflies -> log        pyflies[0.4.2]  Generator for log/debug files.
pyflies -> psychopy   pyflies-psychopy[0.1.1] Generator for generating
                                      PsychoPy code from PyFlies
                                      descriptions
```

Generators can be called by the `textx generator` command as demonstrated in Section 6.6. In the current version of PyFlies, we provide the two built-in generators and the generator for the PsychoPy library that is developed as a separate project. We describe those generators in the next subsections.

5.1. Built-In Generators

The main `PyFlies` project itself registers two general-purpose generators: `CSV` and `Log` (Figure 1).

The `CSV` code generator generates a *Comma-Separated Values* file for the first expanded table in the experiment flow. Due to `repeat` loops and the possibility of having multiple tests, there can be many different tables in the experiment flow. These CSV files can be used as input condition files for other experiment builders.

The `Log` generator generates a textual log file with detailed information about the experiment and its flow. It is important during test development as the test is given in a sort of "compact form", i.e., a condition table expands during run-time as explained in Section 4.3. Furthermore, due to repetitions and randomness, and the way components are instantiated during each trial, it might not be apparent to novice language users what are the actual steps the test will take. The log shows the actual steps that happen in chronological order. Thus, it serves as a sort of debugger which is very helpful during the initial stages of learning the language. It is a plain text file whose content is conveniently indented to take advantage of VS Code default code folding capabilities. Figure 3 shows the usage of a log file to reveal a particular trial of a particular test run.

```
Parity > Parity.pflog
65    | 8       | even    |
66    | 9       | odd     |
67
68    ---- Trials -----
69
70  >   TRIAL 2:1 ···
97
98
99    TRIAL 2:2
100
101     | number | parity |
102     |--------+--------|
103     | 2      | even   |
104
105 >   Variables: ···
111
112   Phases:
113
114     fix:
115       at 0 cross(position (0, 0), size 20, color #ffffff, fillColor #ffffff) for 2304
116
117     exec:
118       at 0 text(content 2, position (0, 0), size 20, color #ffffff, fillColor #ffffff) for 0
119       at 0 keyboard(valid [odd, even], correct even) for 0
```

Figure 3. A snippet of a log file showing the test trial's detail.

All generators are treated the same, so there are no fundamental differences between these two generators provided by the core project and any other generator developed for PyFlies (or for any textX language to be more precise).

5.2. PsychoPy Generator

The first target platform's code generator that is fully developed is the generator for the PsychoPy experiment library. We have chosen this library as our first target due to its maturity, flexibility, and the fact that it is supported on all major operating systems, and can achieve high levels of precision and accuracy [41]. PsychoPy is implemented in Python, which is another positive side as Python has become a de facto standard in scientific research in recent years.

The PsychoPy generator implementation consists of two files. The first file is a Python file (Listing 11) that is used to set up the Jinja template engine and to perform mapping of all PyFlies types and values to PsychoPy. The second file is the Jinja template (Listing 13) that is used to produce the final output experiment as a runnable Python script. The template consists of fixed parts (e.g., lines 1–4) that are written to the target file without change, and variable parts (e.g., lines 8–16) that are expanded by the template engine based

on the information from the model. The target script can be run by either PsychoPy Runner or by the standard Python interpreter.

Listing 13. A part of the Jinja template in the PsychoPy generator.

```
# Create some handy timers
globalClock = core.Clock()
trialClock = core.Clock()
routineTimer = core.CountdownTimer()

{# Execute flow #}
# Experiment flow
{% for inst in m.flow.insts %}
  {% if inst|type == "TestInst" %}
execute_test({{inst.name}}_{{loop.index}}
             {%- if inst.random %}, random=True{% endif %}
             {%- if inst.practice %}, practice=True{% endif %})
  {% else %}
execute_screen({{inst.name}}, {{inst.name}}_{{loop.index}}, {{inst.
    duration|duration}})
  {% endif %}
{% endfor %}

# Flip one final time so any remaining win.callOnFlip()
# and win.timeOnFlip() tasks get executed before quitting
win.flip()
```

A part of Python code produced by the PsychoPy generator for the Eriksen flanker experiment described in Section 6 is given in Listing 14. We can see how the variable parts of the Jinja template from Listing 13 were expanded in the final output code. For the full source code please see the experiment repository on GitHub (https://github.com/pyflies/EriksenFlanker, accessed on 1 July 2021).

Listing 14. A part of the generated PsychoPy code for the Jinja template given in Listing 13.

```
# Create some handy timers
globalClock = core.Clock()
trialClock = core.Clock()
routineTimer = core.CountdownTimer()
# Experiment flow
execute_screen(intro, intro_1, 0.0)
execute_test(EriksenFlanker_2, random=True, practice=True)
execute_screen(real, real_3, 0.0)
execute_test(EriksenFlanker_4, random=True)

# Flip one final time so any remaining win.callOnFlip()
# and win.timeOnFlip() tasks get executed before quitting
win.flip()
```

6. PyFlies Case Study

This section presents a case study on creating a real experiment. It is written in a form of a tutorial so that it could be easy to replicate.

The first step to perform is to install and setup PyFlies. The complete process of PyFlies installation is available in the *Getting Started* video tutorial (getting started video, https://www.youtube.com/watch?v=NVB2JHbCLY0, accessed on 1 July 2021).

For the experiment, we choose a variation of the Eriksen flanker task [42], inspired by the tutorial in [7].

In the Eriksen flanker task, visual stimuli are displayed to subjects instructed to press the corresponding key (e.g., the left key for a left-pointing arrow). In some trials, the surrounding visual field contains stimuli that might be in the same category as the target (e.g., same arrow orientations). These trials are said to be congruent. In other trials, the surrounding stimuli might belong to the opposite category than the target (e.g., opposite arrow orientations), thus flanking the target. These trials are incongruent. It has been

consistently found, in multiple studies, that incongruent flanking items slow down RTs relative to congruent flankers [43].

6.1. Stimuli Image Preparation

Figure 4 shows the four stimuli where (a,b) are congruent, as the central arrow points in the same direction as the surrounding, and (c,d) are incongruent. The arrows usage is not mandatory in this test. We could use other combinations of target and flanking stimuli (letters, numbers, colored shapes, etc.). PyFlies enables easy tests variations.

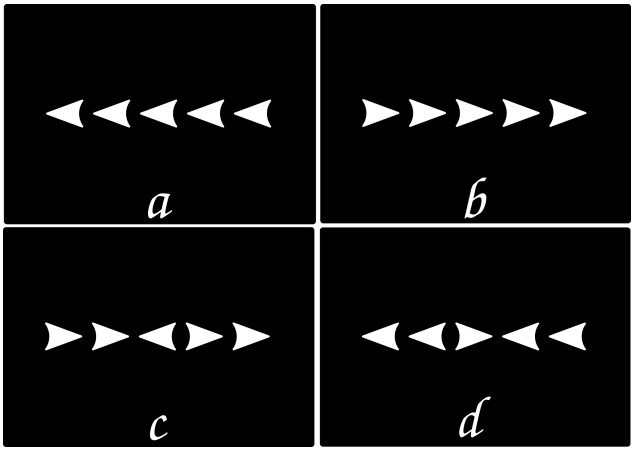

Figure 4. Visual stimuli for the Eriksen flanker task: (**a**–**b**) congruent; (**c**–**d**) incongruent

We recommend Inkscape [44] for image preparations, as a good free and open-source vector drawing program. Inkscape is used for editing images in Scalable Vector graphic format (SVG) but can export to other vector and bitmap formats. We recommend Portable Network Graphics (PNG) for bitmap formats, as it is a lossless format, which provides an alpha channel for transparency.

A source code for the complete example is available at GitHub Section 5.2 so you can obtain the images and save them in the `images` folder of the project with file names of the form `<in>congruent-<left/right>.png`. It is important to have consistent naming as we shall see in the next section.

6.2. Test Definition

In this part of the study, one should go to the *EriksenFlanker* folder on the left side, click the *New File* button and type in the name *eriksen.pf*. It is important to provide the *pf* file extension as it is used by the editor to recognize files as PyFlies specifications.

In the editor, one should provide the code from Listing 15. Instead of just copy/pasting, one should try to type the code to become comfortable the editor. You will notice that PyFlies provides code snippets. For example, when you start typing word `test`, you will be provided with the possibility to expand the test snippet. Press the `TAB` key to expand the snippet. Type the name of the test `EriksenFlanker` and press `TAB` again to go to the next field. Each code snippet can provide multiple fields/locations.

Listing 15. Eriksen flanker test specification.

```
directions = [left, right]
congruencies = [incongruent, congruent]
repeats = 1

test EriksenFlanker {

    | repeat          | direction       | category          |
    | --------------- | --------------- | ----------------- |
    | 1..repeats loop | directions loop | congruencies loop |

```

```
11      fix -> cross() for 1000..3000 choose
12      exec -> image(file "images/{{direction}}-{{category}}.png", size
             100)
13              keyboard(valid directions, correct direction)
14      error -> sound(freq 800) for 300
15      correct -> sound(freq 400) for 300
16  }
```

In lines 7–9 is a condition table with three variables: `repeat`, `direction`, and `category`. The `repeat` variable loops over a range `1..repeats` where `repeats` is a parameter specified during the flow of the experiment (line 8 in Listing 17). It controls the total number of test trials. Our expanded table contains all possible triplets of `repeat`, `direction`, and `category` with a total of `repeats x 2 x 2` number of rows.

In line 11, we specify the `fix` phase as a cross shape displayed with a random duration chosen from the range of `[1000, 3000]` ms.

In the `exec` phase (lines 12–13), an image is displayed with a size set to `100`% and unspecified duration. Since the duration is not specified, the stimulus runs until the end of the phase. In this case, the `keyboard` component that follows controls when the phase is going to finish. The filename of the image given in line `12` has interpolated values `direction` and `category`. These values are provided for each trial by the condition table. For example, a valid file name can be `images/left-incongruent.png`.

The `keyboard` component given in line 13 specifies valid responses as a list of directions. We shall later provide mappings from directions to keyboard keys in the `target` configuration section (Section 6.5). This task has only one correct response, and it is given by the `direction` variable of the current trial.

The last thing to do is to define optional `correct` and `error` phases. Here, we want to give audible feedback to the subject with different frequencies for correct and error responses. For that, we use the `sound` component with given frequencies and the duration set to `300 ms`.

6.3. Screens Definition

To prepare the subject for each batch of trials, we define two screens given in Listing 16. We have an `intro` screen that introduces the test to the subject. Our experiment will show this screen at the beginning before the practice trials run. The `real` screen will be shown after the practice run finish but before real trials start. These screens are plain text without any dynamically rendered parts.

Listing 16. Eriksen flanker test's screen definitions.

```
screen intro {
    Welcome
    -------

    In the following test you will be presented
    with a line of 5 arrows pointing left or right.
    You should respond by the direction of the
    *middle arrow* as fast as possible by pressing
    left or right arrow on the keyboard or
    touching/clicking the appropriate button on
    the screen.

    You will first do a practice run with 4 trials.
    A real run with 20 trial will be performed
    afterwards.

    Press SPACE or touch/click the screen
    to continue
}

screen real {
    Real block
    ----------
```

```
    Now a real block of trials will
    be performed.

    Press SPACE or touch/click the screen
    to continue
}
```

6.4. Flow Definition

Up until this point, we have defined all elements of our experiment. Now we need to define the flow of the execution. In this experiment, the flow is simple: we show the introduction screen, execute practice trials, then show the `real` screen and lastly execute real trials where the data will be collected.

The flow of the experiment is given in Listing 17. In line 2, we use the `show` statement to show the `intro` screen. After the screen ends (terminates by a press on the keyboard), the test is executed in practice mode. Note the parameters of the test: `practice` set to `true`, and `random` set to `true`. It will run the test in practice mode with randomized trials.

Listing 17. Eriksen flanker test's flow definition.

```
1 flow {
2   show intro
3   // Perform a practice test series
4   execute EriksenFlanker(practice true, random true)
5
6   show real
7   // Perform a real test series (collects data)
8   execute EriksenFlanker(repeats 5, random true)
9 }
```

After practice trials end, we display the `real` screen (line 6), giving information to the subject that the real test is about to begin. In line 8, we execute the same `EriksenFlanker` test, but now we do not specify the `practice` parameter so it defaults to `false`. We passed an additional parameter `repeats` set to `5` as we wanted our four trials from the test's condition table to be repeated five times.

6.5. Target Configuration

The last piece of the puzzle is the configuration of the target generator. It is optional, and we can leave it out if the default settings of the target code generator suit our experiment. In our experiment, we use directions in the `keyboard` component. Left and right arrow keys are called `left` and `right` in the target PsychoPy library, so we do not need any mapping.

In case we would like to use other keys and, for example, background color, we could use a configuration similar to the one given in Listing 9.

6.6. Generating and Running the Experiment

At this point, the experiment description is complete. The final step is to produce the runnable program. This is achieved by running the target code generator over the PyFlies model file. Since we have PsychoPy generator installed in our Python virtual environment, we call the generator:

```
$ textx generate eriksen.pf --target psychopy --overwrite
Generating psychopy target from models:
/home/igor/EriksenFlanker/eriksen.pf
Creating /home/igor/EriksenFlanker/eriksen.py
Done. Files created/overwritten/skipped = 1/0/0
```

The `--overwrite` flag instructs the generator to overwrite the target file if it already exists. From the output, we can see that the target Python script `eriksen.py` is produced. This script is our experiment implemented to use the PsychoPy library. We ran this experiment as any other Python script:

```
$ python eriksen.py
```

The experiment will run as instructed in our PyFlies model, and the data from the `real` block of trials will be stored in the `data` folder. The data will contain all the relevant information about each trial.

6.7. More Examples

The source code repository at GitHub has several complete examples (https://github.com/pyflies/pyflies/tree/main/examples, accessed on 1 July 2021). An example of how to create a block of trials and counterbalancing is also provided. There is also an example and a video tutorial for Posner cueing task (https://www.youtube.com/watch?v=Fm_XBnqyGfI, accessed on 1 July 2021) [45] which provide more insight into the process of producing and using tests with PyFlies.

7. Discussion

In this section, we discuss the limitations of the current approach and implementation. We also give some ideas for further development directions and improvements.

7.1. Calling Target Platform Code

For a DSL to be successful, it must be effective and efficient in its intended domain—that is, experiments that are considered to fit the domain should be expressible with the language in an optimal way. This can be expressed as the coverage of the domain [46]. DSL might cover too little of the domain, making some valid experiments impossible to define, or more than needed, making the language unnecessarily complex.

There is also a typical trade-off between generality and completeness. PyFlies language should be general enough to support different target platforms but detailed enough to enable a broader set of features. In other words, PyFlies is limited to a set of features common to possible target experiment platforms.

A usual approach to remedy this issue is to enable extending PyFlies definitions at prescribed places using the target general-purpose language. For example, we could provide means to call Python functions and use Python expressions at specific places in the experiment. Although this would make PyFlies more flexible, as experimenters could utilize a huge ecosystem of Python libraries, it would hamper portability between experiment platforms. Nevertheless, we find that the pragmatism of providing the ability to call into the target platform code outweighs the portability problem, so we plan to support it in the future PyFlies versions.

Another feasible approach is to use PyFlies components, which are abstract enough to enable making components with target-specific semantics. As we have already mentioned before, PyFlies component DSL can be exposed to end-users and generator authors. That would make it possible to use target-specific components in the experiment design.

7.2. Unavailability of PyFlies Features on Target Platforms

Depending on the target platform flexibility, there is always a danger that some PyFlies features cannot be mapped to target platform features, i.e., the feature set of PyFlies is not a subset of the target platform feature set.

In this case, the only option is to warn the experimenter that the feature is not available and that the experiment description should be altered to avoid the feature.

7.3. Pre-Evaluation of PyFlies Expressions

All expressions were pre-evaluated during compilation, and a generator obtained the final values. This is fine for non-random expressions, but random expressions (e.g., `choose` or `shuffle` subexpressions) are the problem as values must be generated at run-time to be truly (pseudo)random. To support the run-time generation of random values, and the ability to call into target-specific code, expressions should be translated to the target platform.

This feature is particularly important for defining timing values such as *inter-stimuli intervals* (ISI) where the user would like to implement a particular approach in choosing random values (e.g., using 50 ms steps so that the ISI is an integer number of 60 Hz screen refreshes, or using a Gaussian distribution of values). This could also be beneficial for custom experiment designs where different randomizations and selection of conditions can be specified.

One way to implement expression mapping is to require each target to supply mapping for each PyFlies type/operation. That might be relaxed to be just a recommendation, in which case PyFlies compiler might pre-calculate all subexpressions which are not available in the target generator. For example, providing just mapping for `choose` would be enough to support random run-time generation in simple cases where only `choose` is used, but for example in `1..10 choose + 10` the target is required to support + operation mapping.

We can execute an analysis of expressions and issue warnings if some part of an expression could be translated but is not due to the non-availability of translation for operations in the other parts of the expression.

Another consideration is in which case expression translation should be used. For example, loop expression for table expansion should stay pre-evaluated to have a stable pre-determined number of trials for a test. Conversely, component parameter values, duration, time reference, etc. could be made translatable.

7.4. Additional Generators

One of the benefits of having DSL with code generators is to achieve experiment portability across a wide range of experiment platforms. For this, code generators for multiple platforms should be implemented. Our current plan is to provide at least one generator for a web-based platform.

In the current version, we have implemented a generator for PsychoPy. One direct way for PyFlies to target the web is to support PsychoPy Builder format as the output (https://www.psychopy.org/psyexp.html, accessed on 1 July 2021). The PsychoPy Builder format is a textual XML-based format from which both Python and JavaScript code can be generated. By implementing the PyFlies code generator for PsychoPy Builder XML target, we can receive both Python and JavaScript support while the target code quality is guaranteed as the code generator is created by PsychoPy experts.

Additionally, as code generators can be implemented as separate independent projects, they can be contributed by other developers who are knowledgeable on particular target platforms.

7.5. Timing Considerations

Since PyFlies is not a run-time platform, the timing accuracy and precision can be as good as what can be achieved by the chosen target platform. Nevertheless, it is still a code generator's responsibility to provide the best timing performance possible with the target experiment platform. One of the improvements for achieving higher precision with visual stimuli, due to the limitation of the display's refresh rate, would be an option to specify durations in ticks instead of milliseconds [47].

An overview of the timing performance of various experiment run-time platforms can be found in [48].

7.6. Better Support in Editors

For the PyFlies language adoption and positive overall user experience, editor support is of great importance. The current VS Code extension offers code snippets and syntax highlighting that helps in the development. However, editor support could be further extended.

First, we could provide better syntax and semantic checks with an explanation of what the user is doing wrong and a suggestion of how the problem might be fixed. Our plan to support this is by implementing *Language Server Protocol* (LSP) [49] for PyFlies.

LSP is an open protocol started by Microsoft to separate language cleverness from the editors and IDEs. This way, a single PyFlies LSP server would contain all the knowledge to perform operations such as auto-complete, go to definition, or documentation on hover. This server would serve multiple editors and development environments, reducing the effort to support new editors.

Another feature that would help writing condition tables is displaying an expanded version of the table in hovering popups.

Although collaborative editing in PyFlies is supported by using text-based version control systems such as git, we could further improve interactivity by implementing a web-based editor. In web-based editing, the experiment is stored and edited collaboratively in the cloud. With the code generator for the web platforms, this would make the whole process available online, lowering the barrier to use the tool. All the user would need, to both author and deploy experiments, is an account on the cloud service.

7.7. Initial Feedback

In further work, we plan to perform an analysis of the language and the approach by performing a controlled experiment with the users. However, we do have some anecdotal evidence on the usability and the general feel of the language that we gathered from our students during their course assignments and from psychologists with whom we were discussing language and its features. We observed that users that are already familiar with some programming language and general code editing were able to pick up the language quickly after seeing our video materials and completing some training from our side. Their general feedback was positive in comparison to their experience in using general-purpose languages. They observed that they were able to specify experiments quicker with fewer errors. However, users that were mostly accustomed to graphical builders experienced a steeper learning curve as they had to become familiar with standard code editing idioms.

8. Related Work

In this section, we present and analyze related work. We organize this section into two subsections. DSLs have been successfully used in various domains. To show the versatility of the approach, the first subsection presents some DSL-based research from different domains and compares them with our work from different viewpoints. The second subsection gives an overview of relevant work in the field of psychology test builders.

8.1. Domain-Specific Languages

Kosar et al. [21] present a unique hand rehabilitation platform *RehabHand* based on DSL and code generation techniques. The language uses a simple textual syntax and enables therapists to write rehabilitation exercises in natural, domain-specific terminology and share them with patients. The exercise is then translated to source code which is uploaded to various rehabilitation devices. The approach regarding the language itself is similar to ours. The semantics of the language is described, similarly to our work, as a source code generator. However, the technology used to develop a source code generator is Xtend [50], a general-purpose programming language for Java Virtual Machine. The language itself and the supporting language services are developed using a Java-based language workbench *xText* [50].

The work of [51] presents a RobotML, a DSL to design, simulate and deploy robotic applications. The language syntax is graphical and the language defines not only abstractions from the domain of robotics, but also component-based architectures. The authors report that, although the development time has not significantly decreased they observed multiple advantages: (a) more time is spent on the design than on dealing with low-level details, (b) the architecture is made explicit, (c) switching to a new target platform is much easier. Similar to our work, this language is meant to be used by domain experts. However, RobotML uses graphical notation while PyFlies uses textual.

Visser presents a case study in DSL engineering in [46]. The author has designed and built *WebDSL*, a technical DSL for web applications. In the development of the language, several DSLs for language engineering have been used. SDF [52] has been used for syntax definition while Stratego/XT [53] has been used for code generation. Since this is a technical language, it is oriented towards software developers.

Johanson and Hasselbring [15] present an empirical study of a non-technical, i.e., application domain DSL, by evaluating the *Sprat Ecosystem DSL* [54], which is a DSL for specifying high-performance marine ecosystem simulation experiments, for its effectiveness and efficiency. The results show that the participants' correctness point score was increased by 61–63% compared with the GPL-based solution and their time spent on the tasks was reduced by 31–56%. Furthermore, the Ecosystem DSL receives higher user ratings than the GPL-based solution concerning quality characteristics such as simplicity of use and maintainability of solutions. DSL analyzed in this study is, similarly to ours, targeted towards scientists.

Pajić et al. [55] present a specification of a domain-specific modeling language for the extraction of event logs from *enterprise resource planning* (ERP) systems. The models defined with this language are automatically validated and transformed into SQL code. Contrary to our solution, the language presented in this work defines a graphical syntax by which the domain experts can create custom complex queries to obtain event logs. Similar to our solution, the language is oriented towards users with little programming skill.

In paper [56], another technical DSL is presented whose aim is to facilitate the development of database-oriented business applications. Similar to our work, the presented language uses textual syntax but it is built using xText [50]. The language is based on *entities* and *services* concepts. From the model specification, a fully functional web application can be generated.

A graph layouting library and DSL for its configuration are presented in [57]. This DSL is a technical declarative language whose aim is to ease the configuration of a complex graph layouting library. It is implemented in textX [33].

Another technical DSL is ALAS [58]. This language is designed to support the development of mobile intelligent agents which are deployed to Siebog multi-agent system [59]. ALAS facilitates interoperability by enabling translation of agents' specifications to target GPL supported by the node where the agent runs.

In the paper [60] a music programming and live coding *HMusic* DSL is presented. This language is built as a so-called *internal DSL* [28]. These kinds of languages are embedded into GPL languages by utilizing features of the host language. *HMusic* is embedded in the Haskell functional programming language. Similar to ours, this language belongs to the application domain category. However, internal languages are constrained by their host language so the syntax cannot be tailored to users' needs. Furthermore, the usage of the language still requires programming skills as the users operate with the host language and its tool-chain.

8.2. Psychology Test Builders

Previously, we introduced a classification of psychological test builders based on the GUI-based, GPL-based, and DSL approaches. In this section, we focus on the GUI-based ones as they naturally operate on a domain-specific level using domain concepts but focusing on graphical syntaxes and the text-based DSL approach which we use in this paper.

8.2.1. GUI-Based

OpenSesame [1] is an established free and open-source software. It is written in the Python programming language and provides a usual GUI-based means for experiment creation. It has a palette of objects and the experiment can be created by drag&drop operation. It provides great flexibility enabling the writing of snippets of Python code where necessary. The stimuli presentation can be delegated to various backends. It is

extensible so additional features may be provided through plugins. Save format can be a plain text which resembles a DSL so it could be used as a basis for text-based version control and collaboration. Although direct editing is possible, OpenSesame files seem too verbose and not optimized for it.

E-Prime [61] is a tool for quickly creating psychological experiments using a palette of objects and drag&drop operations. The tool features millisecond accurate stimulus presentation and response time, various stimuli objects such as sounds, images, videos, text, buttons. It supports integration with various hardware devices. However, E-Prime is not free open-source software and it runs on Windows operating system only.

Lab.js [4] is a browser-based, free and open-source experiment builder. Experiments can be created by writing code using a high-level JavaScript library or using an online graphical interface. Experiments comprise HTML, CSS, and JavaScript files that can be downloaded, run locally, or uploaded to cloud-based services. It directly supports concepts such as screens, flow, page, loop, sequence. Similar to Lab.js is Gorilla [5]. It is also an online browser-based GUI experiment builder with the aim to provide a complete integrated SaaS (*Software as a Service*) solution where users can build, evolve, manage and deploy their experiments.

PsychoPy [3] is another popular free and open-source solution. It started as a GPL-based solution in the Python programming language but in recent years the project team developed a GUI-builder called *PsychoPy Builder* which can be used to quickly create experiments. From the GUI-based design, a code for the PsychoPy library can be generated automatically or the design can be saved as an XML-based save file format. PyFlies in this version provide a source code generator for the PsychoPy library (Section 5.2).

8.2.2. DSL-Based

The most relevant psychology builders to our work are those that use textual DSLs built from scratch. To the best of our knowledge, in this category we have PEBL [8], DMDX [62], and PsyToolkit [63]

The current offering in the area of DSL for psychology tests seems to lack the desired level of abstraction and readability. PEBL [8], while being flexible and capable of providing support for a wide range of experiments, demonstrated by a large battery of tests, is more on the GPL side of the languages' spectrum with concepts similar to those found in other general-purpose programming languages. That makes PEBL specification more verbose and harder to comprehend.

PsyToolkit [63] is probably the most similar to our work. It is not free and open-source but it is free to use. It provides a browser-based editor and DSLs for creating questionnaires and reaction-time experiments. There is a library of free examples which can be downloaded from the project website, investigated, and run through the provided free service. While the questionnaire DSL looks simple and easy to learn, the reaction-time experiments DSL lacks abstraction and ease of use. Contrary to PyFlies, it is defined in a more imperative style and the table of conditions can only be specified in expanded form (i.e., all entries must be provided). This makes experiments unnecessarily verbose.

DMDX [62] is another popular DSL with GUI front-end developed in recent years [64]. DMDX language is large. According to [64], it comprises 414 keywords with 224 synonyms and 75 different branching keywords. The syntax of DMDX is designed more than three decades ago and seems optimized not for humans but machines. That makes DMDX usage hard without a specialized GUI builder.

9. Conclusions

In the last two decades, many interesting tools for building psychology experiments have emerged [1,3–6]. However, these builders are either GUI-based or GPL-based.

GUI-based builders cannot effectively utilize the uptake in text-based distributed collaboration and version control. Furthermore, they might not appeal to users who

become used to modern text editors and text-editing idioms but would like to work on a higher level of abstraction.

On the other hand, GPL-based builders require programming skills and operate on the level of general-purpose programming concepts. This hinders the readability and comprehensibility as the experimental design is not apparent by looking at the code.

To the best of our knowledge, very few builders which belong to the DSL crowd are either closer to general-purpose languages (PEBL [8]), too low-level and not optimized for domain expert's usage (DMDX [62]), or could be improved in terms of abstraction and conciseness (PsyToolkit [63]).

As a possible move in the right direction, this paper presented PyFlies, a free and open-source domain-specific language for the specification of experiments in psychology. PyFlies is used for several years in educational settings. However, we believe that it could be useful for psychology practitioners as well.

Our aim with the language was to capture the experiments' essence in a concise and readable form, without the technical clutter, usually introduced by target execution platforms.

To support a gentle learning curve, we provide several examples, full documentation, and a video tutorial series.

We provide a Visual Studio Code extension for a better user experience, especially in the condition tables editing domain. The extension provides code snippets and syntax highlighting, which makes the authoring of new experiments a pleasant activity. Further work on the editor is targeted towards better semantic checks with explanations, automatic code fixes, Language Server Protocol support, web-oriented editing, etc.

The PyFlies language is based on Python programming language and features a modular architecture where generators can be developed as separate projects. In this early version, we provide a code generator for the PsychoPy target. We plan to develop code generators for other platforms as well.

We are aware that PyFlies has limitations. In the previous section, we discussed the current shortcomings of the approach and implementation and gave some ideas for further work we plan to do. Some limitations are inherent due to the DSL approach, whereas the others are current implementation issues that will be improved in future versions.

In further work, we plan to perform an analysis of the language and the approach by performing a controlled experiment with the users where we would measure the time taken to build a test, difficulties in using the language, and VS Code extension. We also plan to carry out a language usability evaluation using some of the established conceptual frameworks (e.g., Use-Me [65]).

PyFlies is a free and open-source project which is developed by the community. It is hosted at GitHub Section 1, and it is provided under the terms of the GPL 3.0 license. Everyone is welcome to contribute code, documentation, tests, bug reports, etc. We hope that this paper will motivate researchers with programming experience, who are knowledgeable in experimental software, to implement additional code generators.

Author Contributions: Conceptualization, I.D. and M.D.; methodology, I.D. and M.D.; software, I.D.; validation, M.D., I.D., J.V. and S.N.; investigation, M.D., I.D., J.V. and S.N.; writing—original draft preparation, I.D. and M.D.; writing—review and editing, J.V. and S.N.; visualization, J.V. and S.N.; supervision, I.D. and M.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: PyFlies source code, examples, documentation and video materials are available at https://github.com/pyflies/, accessed on 1 July 2021. Additional information is available from corresponding authors upon a reasonable request.

Acknowledgments: We wish to thank Michael MacAskill for providing valuable feedback on PyFlies via the PsychoPy forum. We also thank the students from the University of Novi Sad and the University of Pristina-Kosovska Mitrovica who provided feedback and reported bugs while using PyFlies as a part of their course assignments.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Mathôt, S.; Schreij, D.; Theeuwes, J. OpenSesame: An open-source, graphical experiment builder for the social sciences. *Behav. Res. Methods* **2012**, *44*, 314–324. [CrossRef] [PubMed]
2. Schneider, W.; Eschman, A.; Zuccolotto, A. *E-Prime User's Guide*; Psychology Software Tools, Inc.: Pittsburgh, PA, USA, 2002.
3. Peirce, J.; Gray, J.R.; Simpson, S.; MacAskill, M.; Höchenberger, R.; Sogo, H.; Kastman, E.; Lindeløv, J.K. PsychoPy2: Experiments in behavior made easy. *Behav. Res. Methods* **2019**, *51*, 195–203. [CrossRef] [PubMed]
4. Henninger, F.; Shevchenko, Y.; Mertens, U.; Kieslich, P.J.; Hilbig, B.E. Lab.js: A Free, Open, Online Study Builder. *Behav. Res. Methods* **2021**. [CrossRef] [PubMed]
5. Anwyl-Irvine, A.L.; Massonnié, J.; Flitton, A.; Kirkham, N.; Evershed, J.K. Gorilla in our midst: An online behavioral experiment builder. *Behav. Res. Methods* **2020**, *52*, 388–407. [CrossRef] [PubMed]
6. Krause, F.; Lindemann, O. Expyriment: A Python library for cognitive and neuroscientific experiments. *Behav. Res. Methods* **2014**, *46*, 416–428. [CrossRef] [PubMed]
7. de Leeuw, J.R. jsPsych: A JavaScript library for creating behavioral experiments in a Web browser. *Behav. Res. Methods* **2014**, *47*, 1–12. [CrossRef] [PubMed]
8. Mueller, S.T.; Piper, B.J. The psychology experiment building language (PEBL) and PEBL test battery. *J. Neurosci. Methods* **2014**, *222*, 250–259. [CrossRef] [PubMed]
9. Chacon, S.; Straub, B. *Pro Git*, 2nd ed.; Apress: New York, NY, USA, 2014.
10. O'Sullivan, B. *Mercurial: The Definitive Guide*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2009
11. Voelter, M.; Benz, S.; Dietrich, C.; Engelmann, B.; Helander, M.; Kats, L.C.; Visser, E.; Wachsmuth, G. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*; M.Voelter/dslbook.org: Berlin, Germany, 2013.
12. Mernik, M.; Heering, J.; Sloane, A. When and how to develop domain-specific languages. *ACM Comput. Surv. (CSUR)* **2005**, *37*, 316–344. [CrossRef]
13. Kosar, T.; Mernik, M.; Carver, J.C. Program Comprehension of Domain-specific and General-purpose Languages: Comparison Using a Family of Experiments. *Empir. Softw. Eng.* **2012**, *17*, 276–304. [CrossRef]
14. Kosar, T.; Gaberc, S.; Carver, J.C.; Mernik, M. Program Comprehension of Domain-Specific and General-Purpose Languages: Replication of a Family of Experiments Using Integrated Development Environments. *Empir. Softw. Eng.* **2018**, *23*, 2734–2763. [CrossRef]
15. Johanson, A.N.; Hasselbring, W. Effectiveness and Efficiency of a Domain-Specific Language for High-Performance Marine Ecosystem Simulation: A Controlled Experiment. *Empir. Softw. Eng.* **2016**, *22*, 2206–2236. [CrossRef]
16. Fowler, M. Language Workbenches: The Killer-App for Domain Specific Languages. 2005. Available online: http://www.martinfowler.com/articles/languageWorkbench.html (accessed on 1 July 2021).
17. Fischer, G.; Giaccardi, E.; Ye, Y.; Sutcliffe, A.G.; Mehandjiev, N. Meta-design: A manifesto for end-user development. *Commun. ACM* **2004**, *47*, 33–37. [CrossRef]
18. van Deursen, A.; Visser, J. Domain-specific languages: An annotated bibliography. *ACM Sigplan Not.* **2000**, *35*, 26–36. [CrossRef]
19. Schmidt, D.C. Model-driven engineering. *Comput.-IEEE Comput. Soc.* **2006**, *39*, 25. [CrossRef]
20. Da Silva, A.R. Model-driven engineering: A survey supported by the unified conceptual model. *Comput. Lang. Syst. Struct.* **2015**, *43*, 139–155.
21. Kosar, T.; Lu, Z.; Mernik, M.; Horvat, M.; Črepinšek, M. A Case Study on the Design and Implementation of a Platform for Hand Rehabilitation. *Appl. Sci.* **2021**, *11*, 389. [CrossRef]
22. Dmitriev, S. Language Oriented Programming: The Next Programming Paradigm. JetBrains, 2004. Available online: https://resources.jetbrains.com/storage/products/mps/docs/Language_Oriented_Programming.pdf (accessed on 1 July 2021).
23. Kleppe, A. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*; Addison-Wesley: Boston, MA, USA, 2009
24. Bentley, J. Programming pearls: Little languages. *Commun. ACM* **1986**, *29*, 711–721. [CrossRef]
25. Sprinkle, J.; Mernik, M.; Tolvanen, J.; Spinellis, D. What Kinds of Nails Need a Domain-Specific Hammer. *IEEE Softw.* **2009**, *26*, 15–18. [CrossRef]
26. Van Deursen, A.; Klint, P. Little Languages: Little Maintenance? *J. Softw. Maint. Res. Pract.* **1998**, *10*, 75–92. [CrossRef]
27. Voelter, M.; Kolb, B.; Birken, K.; Tomassetti, F.; Alff, P.; Wiart, L.; Wortmann, A.; Nordmann, A. Using language workbenches and domain-specific languages for safety-critical software development. *Softw. Syst. Model.* **2019**, *18*, 2507–2530. [CrossRef]
28. Fowler, M. *Domain-Specific Languages*, 1st ed.; Addison-Wesley Professional: Boston, MA, USA, 2010.
29. Eysholdt, M.; Behrens, H. Xtext: Implement your language faster than the quick and dirty way. In Proceedings of the ACM international Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, Reno/Tahoe, NV, USA, 17–21 October 2010; pp. 307–309.

30. Kats, L.; Visser, E. The spoofax language workbench: Rules for declarative specification of languages and IDEs. In *ACM Sigplan Notices*; ACM: New York, NY, USA, 2010; Volume 45, pp. 444–463
31. Klint, P.; Van Der Storm, T.; Vinju, J. Rascal: A domain specific language for source code analysis and manipulation. In Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, Edmonton, AB, Canada, 20–21 September 2009; pp. 168–177.
32. Voelter, M. Language and IDE Modularization and Composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV. GTTSE 2011*; Lecture Notes in Computer Science; Lämmel, R., Saraiva, J., Visser, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7680, pp. 383–430. [CrossRef]
33. Dejanović, I.; Vaderna, R.; Milosavljević, G.; Vuković, Ž. TextX: A Python tool for Domain-Specific Languages implementation. *Knowl.-Based Syst.* **2017**, *115*, 1–4. [CrossRef]
34. Dejanović, I.; Milosavljević, G.; Vaderna, R. Arpeggio: A flexible PEG parser for Python. *Knowl.-Based Syst.* **2016**, *95*, 71–74. [CrossRef]
35. Kosar, T.; Barrientos, P.A.; Mernik, M. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.* **2008**, *50*, 390–405. [CrossRef]
36. Ford, B. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *ACM Sigplan Not.* **2004**, *39*, 111–122. [CrossRef]
37. Dehaene, S.; Bossini, S.; Giraux, P. The mental representation of parity and number magnitude. *J. Exp. Psychol. Gen.* **1993**, *122*, 371. [CrossRef]
38. Schulte, E.; Davison, D. Active documents with org-mode. *Comput. Sci. Eng.* **2011**, *13*, 66–73. [CrossRef]
39. Armin Ronacher and Contributors. Jinja Template Engine. Available online: https://jinja.palletsprojects.com/ (accessed on 10 February 2021).
40. Syriani, E.; Luhunu, L.; Sahraoui, H. Systematic mapping study of template-based code generation. *Comput. Lang. Syst. Struct.* **2018**, *52*, 43–62. [CrossRef]
41. Garaizar, P.; Vadillo, M.A. Accuracy and precision of visual stimulus timing in PsychoPy: No timing errors in standard usage. *PLoS ONE* **2014**, *9*, e112033. [CrossRef]
42. Eriksen, B.A.; Eriksen, C.W. Effects of noise letters upon the identification of a target letter in a nonsearch task. *Percept. Psychophys.* **1974**, *16*, 143–149. [CrossRef]
43. Fox, E. Negative priming from ignored distractors in visual selection: A review. *Psychon. Bull. Rev.* **1995**, *2*, 145–173. [CrossRef]
44. Bah, T. *Inkscape: Guide to a Vector Drawing Program*; Prentice Hall Press: Englewood Cliffs, NJ, USA, 2007.
45. Posner, M.I. Orienting of attention. *Q. J. Exp. Psychol.* **1980**, *32*, 3–25. [CrossRef]
46. Visser, E. WebDSL: A Case Study in Domain-Specific Language Engineering. In *Generative and Transformational Techniques in Software Engineering II*; Lecture Notes in Computer Science; Lämmel, R., Visser, J., Saraiva, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5235, pp. 291–373. [CrossRef]
47. Garaizar, P.; Vadillo, M.A.; López-de Ipiña, D.; Matute, H. Measuring software timing errors in the presentation of visual stimuli in cognitive neuroscience experiments. *PLoS ONE* **2014**, *9*, e85108. [CrossRef] [PubMed]
48. Bridges, D.; Pitiot, A.; MacAskill, M.R.; Peirce, J.W. The timing mega-study: Comparing a range of experiment generators, both lab-based and online. *PeerJ* **2020**, *8*, e9414. [CrossRef] [PubMed]
49. Microsoft. The Language Server Protocol (LSP). Available online: https://microsoft.github.io/language-server-protocol/ (accessed on 10 February 2021).
50. Bettini, L. *Implementing Domain-Specific Languages with Xtext and Xtend*; Packt Publishing Ltd.: Birmingham, UK, 2016
51. Dhouib, S.; Kchir, S.; Stinckwich, S.; Ziadi, T.; Ziane, M. Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 149–160.
52. de Souza Amorim, L.E.; Visser, E. Multi-purpose Syntax Definition with SDF3. In *International Conference on Software Engineering and Formal Methods*; Springer: Cham, Switzerland, 2020; pp. 1–23.
53. Visser, E. Program transformation with Stratego/XT. In *Domain-Specific Program Generation*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 216–238.
54. Johanson, A.; Hasselbring, W. *Sprat: Hierarchies of Domain-Specific Languages for Marine Ecosystem Simulation Engineering*. In *Proceedings of the Symposium on Theory of Modeling and Simulation (TMS/DEVS), Spring Simulation Multi-Conference (SpringSim 2014)*, SCS, Tampa, FL, USA, 13–16 April 2014; pp. 187–192.
55. Pajić Simović, A.; Babarogić, S.; Pantelić, O.; Krstović, S. Towards a Domain-Specific Modeling Language for Extracting Event Logs from ERP Systems. *Appl. Sci.* **2021**, *11*, 5476. [CrossRef]
56. Dejanović, I.; Milosavljević, G.; Perišić, B.; Tumbas, M. A Domain-Specific Language for Defining Static Structure of Database Applications. *Comput. Sci. Inf. Syst.* **2010**, *7*, 409–440. [CrossRef]
57. Vaderna, R.; Vuković, Ž.; Dejanović, I.; Milosavljević, G. Graph Drawing and Analysis Library and Its Domain-Specific Language for Graphs' Layout Specifications. *Sci. Program.* **2018**, *2018*, 7264060. [CrossRef]
58. Sredojević, D.; Vidaković, M.; Ivanović, M. ALAS: Agent-oriented domain-specific language for the development of intelligent distributed non-axiomatic reasoning agents. *Enterp. Inf. Syst.* **2018**, *12*, 1058–1082. [CrossRef]

59. Mitrović, D.; Ivanović, M.; Vidaković, M.; Budimac, Z. The Siebog multiagent middleware. *Knowl.-Based Syst.* **2016**, *103*, 56–59. [CrossRef]
60. Du Bois, A.R.; Ribeiro, R.G. HMusic: A domain specific language for music programming and live coding. In Proceedings of the NIME'19, Federal University of Rio Grande do Sul, Porto Alegre, Brazil, 3–6 June 2019; pp. 381–386.
61. Spapé, M.; Verdonschot, R.; Van Steenbergen, H. *The E-Primer: An Introduction to Creating Psychological Experiments in E-Prime*; Leiden University Press (LUP): Leiden, The Netherlands, 2019.
62. Forster, K.I.; Forster, J.C. DMDX: A Windows display program with millisecond accuracy. *Behav. Res. Methods Instrum. Comput.* **2003**, *35*, 116–124. [CrossRef]
63. Stoet, G. PsyToolkit: A novel web-based method for running online questionnaires and reaction-time experiments. *Teach. Psychol.* **2017**, *44*, 24–31. [CrossRef]
64. Garaizar, P.; Reips, U.D. Visual DMDX: A web-based authoring tool for DMDX, a Windows display program with millisecond accuracy. *Behav. Res. Methods* **2015**, *47*, 620–631. [CrossRef] [PubMed]
65. Barišić, A.; Amaral, V.; Goulão, M. Usability Driven Dsl Development with Use-Me. *Comput. Lang. Syst. Struct.* **2018**, *51*, 118–157. [CrossRef]