TPLP **23** (5): 965–985, 2023. © The Author(s), 2022. Published by Cambridge University Press.
doi:10.1017/S1471068422000084 First published online 14 March 2022

ValAsp: A Tool for Data Validation in Answer Set Programming[*]

MARIO ALVIANO, CARMINE DODARO and ARNEL ZAMAYLA
Department of Mathematics and Computer Science, University of Calabria, Via P. Bucci, cubo 30B, 87036, Rende (CS), Italy
(*e-mails:* alviano@mat.unical.it, dodaro@mat.unical.it, zamayla@mat.unical.it)

submitted 20 February 2021; revised 31 January 2022; accepted 22 February 2022;

Abstract

The development of complex software requires tools promoting fail-fast approaches, so that bugs and unexpected behavior can be quickly identified and fixed. Tools for data validation may save the day of computer programmers. In fact, processing invalid data is a waste of resources at best, and a drama at worst if the problem remains unnoticed and wrong results are used for business. Answer Set Programming (ASP) is not an exception, but the quest for better and better performance resulted in systems that essentially do not validate data. Even under the simplistic assumption that input/output data are eventually validated by external tools, invalid data may appear in other portions of the program, and go undetected until some other module of the designed software suddenly breaks. This paper formalizes the problem of data validation for ASP programs, introduces a language to specify data validation, and presents VALASP, a tool to inject data validation in ordinary programs. The proposed approach promotes fail-fast techniques at coding time without imposing any lag on the deployed system if data are pretended to be valid. Validation can be specified in terms of statements using YAML, ASP and Python. Additionally, the proposed approach opens the possibility to use ASP for validating data of imperative programming languages.

KEYWORDS: Answer Set Programming, data validation, secure coding, fail-fast

1 Introduction

A popular Latin saying starts with *errare humanum est* (translated, to err is human), and clarifies how making mistakes is part of human nature. Computer programmers, being humans, are inclined and not indifferent to errors (Ko and Myers 2003; 2005). Whether a typo in notation, a misspelled word, or a wrong or fragile representation of data, errors in source code files may result in substantial delays in developing an application. Even worse, errors may stay unknown for a long time, until something happens that stimulates the error to cause a crash of the system or some other unwanted and unexpected behavior. In the worst scenario, unknown errors may lead to wrong results that are used to take some business decision, which in turn may ruin a company. (Refer to the paper by Natella *et al.* 2020 for examples of typical errors in software systems.)

[*] This work was partially supported by projects PRIN “Declarative Reasoning over Streams” (CUP: H24I17000080001), PON-MISE MAP4ID “Multipurpose Analytics Platform 4 Industrial Data” (CUP: B21B19000650008), lab LAIA (part of SILA), and GNCS-INdAM.

Fail-fast systems are designed to break as soon as an unexpected condition is detected (refer to the paper by Padhye and Sen (2019) for an example of fail-fast type checking in Java). As often it is the case, the idea is not limited to computer programming, and definitely not originated in software design. For example, many electrical devices have fail-fast mechanisms to provide overcurrent protection – it is better to melt an inexpensive fuse, than to burn a control board. Even if technically an electrical device operating with a fuse replaced by a wire works as expected in most cases, no professional electrician would suggest to tamper with the system in this way. In fact, in case the replaced fuses melt again and again, it is usually evidence that the system has some malfunction that must be detected and fixed.

Computer programming should follow a similar fail-fast approach. Errors must be detected as soon as possible, and reported to the programmer in a non-skippable way, so that the malfunction can be quickly detected and fixed. Data validation is the process of ensuring that data conform to some specification, so that any process in which they are involved can safely work under all of the assumptions guaranteed by the specification. In particular, immutable objects are usually expected to be validated on creation, so that their consistency can be safely assumed anywhere in the system – this way, in fact, an invalid immutable object simply cannot exist because its construction would fail. Guaranteeing validity of mutable objects is usually more difficult and tedious, and almost impossible if there is no invariant on the validity of immutable objects that are part of the mutable objects. (Refer to the papers by (Johnsson *et al.* 2019) and (Vernon 2016) for details on how to tackle the complexity of a business domain in terms of domain primitives and entities.)

Answer Set Programming (ASP; Gelfond and Lifschitz 1991; Niemelä 1999; Marek and Truszczyński 1999) should not be an exception when it comes to errors. However, the language offers very little in terms of protection mechanisms. No static or dynamic type checking are available, and programmers can rely on a very limited set of primitive types, namely integers, strings and alphanumeric constants, with no idiomatic way to enforce that the values of an argument must be of one of these types only – refer to the paper by Calimeri *et al.* (2020) for details of the ASP-Core-2 format. More structured data types are usually represented by uninterpreted function symbols (Lierler and Lifschitz 2009; Eiter and Simkus 2009; Baselice and Bonatti 2010; Calimeri *et al.* 2011; Alviano *et al.* 2010), but again there is no idiomatic way to really validate such structures. Similarly, there is no idiomatic way to specify that input relations satisfy some properties (often expressed in comments, to document the usage of ASP encodings). Even *integrity constraints* may be insufficient to achieve a reliable data validation, as they are cheerfully satisfied if at least one of their literals is false; in fact, integrity constraints are very convenient for discarding unwanted solutions, but not very effective in guaranteeing data integrity – invalid data in this case may lead to discarding some wanted solution among thousands or more equally acceptable solutions, a very hard-to-spot unexpected outcome.

The lack of data validation in ASP is likely due to the quest for better and better performance. After a significant effort to optimize the search algorithms that are one of the main reasons of the success of ASP systems and solvers like CLINGO (Gebser *et al.* 2018), DLV (Leone *et al.* 2019) and WASP (Dodaro *et al.* 2011), to sacrifice a few machine instructions just to validate data that are almost always valid sounds like a blasphemy.

Someone may argue that ASP is not intended to be a general purpose programming language, and therefore input and output are eventually validated by external tools. However, this is a very simplistic assumption, and invalid data may appear in other portions of the program, causing the loss of otherwise acceptable solutions. Everyone is free to follow their own path, but at some point in their life, perhaps after spending hours looking for a typo in the name of a function, any programmer will regret not having had an idiomatic way to specify the format of their data. Quoting the Latins, *errare humanum est, perseverare autem diabolicum* – to err is human, but to persist (in error) is diabolical.

This paper aims at rescuing ASP programmers from some problems due to data validation by proposing a framework called VALASP, written in Python and available at `https://github.com/alviano/valasp`. Specifically, a first contribution of this paper is the formalization of the problem of data validation for ASP programs by combining ASP rules with Python exceptions (Section 3.1). A second contribution of this paper is a language based on the YAML serialization format to specify data validation for ASP programs (Section 3.2), and its compilation into Python code that can be processed by the ASP system CLINGO (Section 3.3). The proposed approach is to specify the format of some data of interest, leaving open the possibility to work with unspecified data types (Section 3). Moreover, the specification can be separated from the ASP program, and the fail-fast approach is achieved by injecting constraints that are guaranteed to be immediately satisfied when grounded, unless data are found to be invalid and some exception is raised. Such a behavior is obtained thanks to *interpreted functions*, briefly recalled in Section 2, where their use for data validation is also hinted. Finally, a few use cases are discussed in Section 4, among them the possibility to take advantage of ASP declarativity for validating complex Python data structures, and related work from the literature is discussed in Section 5 – in particular, differences with sort typed systems like IDP (Cat *et al.* 2018) and SPARC (Marcopoulos and Zhang 2019).[1]

This paper extends a previous work presented at PADL 2021 (Alviano *et al.* 2021) by providing more detailed examples of the implemented approach, by presenting additional use cases of the proposed validation framework, by showing how VALASP can be used to validate Python data structures, and by reporting an extended empirical assessment.

2 Background

ASP programs are usually evaluated by a two-steps procedure: first, object variables are eliminated by means of *intelligent grounding* techniques, and after that stable models of the resulting propositional program are searched by means of sophisticated non-chronological backtracking algorithms. Details of this procedure, as well as on the syntax and semantics of ASP, are out of the scope of this work. Therefore, this section only recalls the minimal background required to introduce the concepts presented in the next sections.

ASP is not particularly rich in terms of primitive types, and essentially allows for using integers and (double-quoted) strings. (We will use the syntax of CLINGO Gebser *et al.*

[1] An extended abstract of this work was presented at the International Conference on Logic Programming (ICLP) 2020 (Alviano and Dodaro 2020).

2018.) More complex types, as for example dates or decimal numbers, can be represented by means of (non-interpreted) functions, or by the so called *@-terms*; in the latter case, the @-term must be associated with a function (written in an imperative programming language like Python) mapping different objects to different symbols in a repeatable way – for example, by populating a table of symbols or by using a natural enumeration.

Example 2.1 (*Primitive types and @-terms*)
Dates can be represented by strings, functions (tuples as a special case) or @-terms, among other possibilities. Hence, `"1983/09/12"`, `date(1983,9,12)` and `@date(1983,9,12)` can all represent the date 12 Sep. 1983, where the @-term is associated with the following Python code:

```
def date(year, month, day):
    res = datetime.datetime(year.number, month.number, day.number)
    return int(res.timestamp())
```

Each representation comes with pros and cons, discussed later in Section 3. ■

Intelligent grounding may process rules in several orders, and literals within a rule can also be processed according to different orderings. A safe assumption made here is that all object variables of an @-term must be already bound to some ground term before the grounder can call the associated (Python) function.

Example 2.2 (*@-term invocation*)
Consider the following program:

```
birthday(sofia, date(2019,6,25)).
birthday(bigel, date(1982,123)).  % Oops! I missed a comma, but where?!?
:- birthday(Person,Date), @is_triple_of_integer(Date) != 1.
```

The Python function associated with the @-term is called two times, with arguments `date(2019,6,25)` and `date(1982,123)`, so that some invariant can be enforced on the second argument of every instance of `birthday/2`. ■

Data validation is the process of ensuring that data conform to some specification, so that any process in which they are involved can safely work under all of the assumptions guaranteed by the specification. Data can be found invalid because of an expected error-prone source (for example, user input from a terminal), or due to an unexpected misuse of some functionality of a system (this is usually the case with bugs). While in the first case it is natural to ask again for the data, in the second case failing fast may be the only reasonable choice, so that the problem can be noticed, traced, and eventually fixed. The fail-fast approach is particularly helpful at coding time, to avoid bug hunting at a later time, but it may also pay off after deployment if properly coupled with a recovery mechanism (for example, restart the process).

Example 2.3 (*Data validation*)
The @-term from Example 2.2 can be associated with the following Python code:

```
def is_triple_of_integer(value):
    if value.type != Function: raise ValueError('wrong type')
    if value.name != 'date': raise ValueError('wrong name')
    if len(value.arguments) != 3: raise ValueError('not a triple')
    if any(arg for arg in value.arguments if arg.type != Number):
        raise ValueError('arguments must be integers')
    return 1
```

Indeed, the presence of `birthday(bigel, date(1982,123))` will be noticed because of abrupt termination of the grounding procedure. Adopting a fail-fast approach is the correct choice in this case, and any attempt of sanification is just a dangerous speculation on the invalid data – should it be `date(1982,1,23)`, or `date(1982,12,3)`? ■

3 A data validation framework for ASP

Data validation can be used in ASP programs thanks to @-terms. However, the resulting code is likely to be less readable due to aspects that are not really the focus of the problem aimed to be addressed. We will illustrate our proposal to accomplish data validation without cluttering an ASP encoding in this section. First, the problem of data validation for ASP programs is formalized in Section 3.1, and a few minimal examples are provided. After that, a language based on the YAML serialization format is introduced in Section 3.2 to specify data validation for ASP programs. Finally, Section 3.3 illustrates how the YAML format is compiled into Python code that can be processed by the ASP system CLINGO.

3.1 Data validation for ASP programs

Let us fix a set $\mathscr{R}$ of *predicate and function names* (or *symbols*), a set $\mathscr{U}$ of *field names*, and a set $\mathscr{T} = \{$`Integer`, `String`, `Alpha`, `Any`$\}$ of *primitive types*. Each type is associated with a set of *facets*, or restrictions. The facets of `Integer` are `enum` to specify a list of acceptable values, `min` (by default -2^{31}) and `max` (by default $2^{31}-1$) to specify (inclusive) bounds, and finally `count`, `sum+` and `sum-` to specify bounds on the number of values, the sum of positive values and negative values. The facets of `String` and `Alpha` are `enum` and `count` as before, `min` and `max` to bound the length, and `pattern` to specify a regular expression. Other types have only the facet `count`.

A *user-defined symbol* s is any name in $\mathscr{R}$. A *field declaration* is a tuple of the form (f, t, F), where f is a field name in $\mathscr{U}$, t is a type in $\mathscr{T}$ or a type defined by the user (i.e. a user-defined symbol), and F is a set of facets for t. A *field comparison* is an expression of the form $f \odot f'$, where $\odot$ is a comparison operator among `==`, `!=`, `<`, `<=`, `>=`, and `>`.

A *user definition* is a tuple of the form (s, D, H, b, c, a), where s is a user-defined symbol, D is a set of field declarations, H is a set of field comparisons (also called *having properties*), and b, c, a are code blocks to be executed respectively before grounding, after the creation of an instance of the user-defined symbol, and after grounding. A *data validation specification* is a tuple of the form (P, A, U), where P is a code block, A is an ASP program, and U is a set of user definitions.

Example 3.1 (*Validation of dates*)
Let `date` be a ternary predicate representing a valid date, and `bday` be a binary predicate whose arguments represent a person and a date. A user definition u_{date} of `date` could be `(date, {(year, Integer, ∅), (month, Integer, ∅), (day, Integer, ∅)}, ∅, ∅,` c`, ∅)`, where c is the following (Python) code block:

```
datetime.datetime(self.year, self.month, self.day).
```

A user definition u_{bday} of `bday` could be `(bday, {(name, Alpha, ∅), (date, date, ∅)}, ∅, ∅, ∅, ∅)`. A data validation specification could be the triple $(P, \emptyset, \{u_{date}, u_{bday}\})$ where P is the (Python) code block

```
import datetime
```

■

Example 3.2 (*Ordering of elements*)
Let `ordered_triple` be a ternary predicate representing a triple of integers in descendent order. A user definition could be `(ordered_triple, {(first, Integer, ∅), (second, Integer, ∅), (third, Integer, ∅)}, {first < second, second < third}, ∅, ∅, ∅)`. ■

Example 3.3 (*Overflow on integers*)
Let `income` be a binary predicate representing incomes of companies, which are summed up in an ASP program. A user definition of `income` could be u_{income} := `(income, {(company, String, ∅), (amount, Integer, {min: 0, sum+: 2147483647})}, ∅, ∅, ∅, ∅)`, specifying that valid amounts are nonnegative and their sum must not overflow. A data validation specification could be $(\emptyset, \emptyset, \{u_{income}\})$. ■

Example 3.4 (*Validation of complex aggregates*)
Consider the constraint

```
:- bound(MAX), #sum{B-B',R : init_budget(R,B), budget_spent(R,B')} > MAX.
```

It bounds the total amount of residual budget, for example for researchers involved in a project. The above constraint can be part of a broader ASP program where `budget_spent/2` depends on some guess on resources and services to purchase. The aggregate above may overflow, and we are interested in detecting such cases and stopping the computation on such unreliable data. To this aim, we can introduce auxiliary predicates in the ASP program A of a data validation specification $(\emptyset, A, U)$:

```
residual_budget(B-B',R) :- init_budget(R,B), budget_spent(R,B').
```

Hence, we can provide a user definition `(residual_budget, {(value, Integer, {min: 0, sum+: 2**31-1}), (id_res, Integer, {min: 0})}, ∅, ∅, ∅)` in U. ■

Specification application. Given an ASP program Π, and a data validation specification (P, A, U), the application of (P, A, U) to Π amounts to the following computational steps:

1. The code block P is executed.
2. For all $(s, D, H, b, c, a) \in U$, the code block b is executed.
3. The ASP program $\Pi \cup A$ is grounded.
4. For all produced instances of a predicate s such that $(s, D, H, b, c, a) \in U$, all types and facets in D and all field comparisons in H are checked, and the code block c is executed. If a check fails, an exception is raised.
5. For all $(s, D, H, b, c, a) \in U$, the code block a is executed.

Example 3.5
The application of the data validation specification from Example 3.1 to an ASP program whose intelligent grounding produces `bday(bigel, date(1982,123))` raises an exception due to the wrong type of the second argument, that is, function `date` is expected to have arity 3, but only 2 arguments are found.

The application of the data validation specification from Example 3.3 to an ASP program comprising facts `income("Acme ASP",1500000000)` and `income("Yoyodyne YAML",1500000000)` raises an exception due to the facet `sum+: 2147483647` of `amount`. This way, an overflow is prevented, for example in

```
total(T) :- T = #sum{A,C : income(C,A)}.
```

which would otherwise produce `total(-1294967296)` in CLINGO and DLV. ■

3.2 A YAML language for data validation

YAML is a human friendly data serialization standard, whose syntax is well-suited for materializing the notion of data validation specification provided in Section 3.1. The YAML files processed by our framework are essentially dictionaries associating keys to other dictionaries, values, and lists. The key `valasp` is reserved, and cannot be used as a symbol or field name. Finally, code blocks are written in Python.

More in details, a data validation specification (P, A, U) is represented by a YAML file comprising the following lines:

```
valasp:
    python: |+
        <Python code block P>
    asp: |+
        <ASP program A>
```

and a block of lines for each user definition (s, D, H, b, c, a):

```
s:
    <field declarations D>
    valasp:
        having:
            - <field comparison h_1>
            - ...
            - <field comparison h_n>
        before_grounding: |+
            <Python code block b>
        after_init: |+
            <Python code block c>
        after_grounding: |+
            <Python code block a>
```

Above, $h_1, \ldots, h_n$ are the field comparisons in H (for some $n \geq 0$), and a field declaration (f, t, F) is represented by

```
    f:
        type: t
        <facets F>
```

where facets are written as key-value pairs.

Example 3.6
Below is a YAML file to validate predicate `bday` of Example 3.1.

```
valasp:
    python: |+
        import datetime
```

```
date:
    year: Integer
    month: Integer
    day: Integer

    valasp:
        after_init: |+
            datetime.datetime(self.year, self.month, self.day)

bday:
    name: Alpha
    date: date
```

The following, instead, is a YAML file to validate predicate `ordered_triple` of Example 3.2:

```
ordered_triple:
        first: Integer
        second: Integer
        third: Integer

        valasp:
                having:
                        - first < second
                        - second < third
```

Note that YAML lists can be written as multiple lines starting with a dash, or in square brackets. Regarding predicate `income` of Example 3.3, its YAML file is the following:

```
income:
    company: String
    amount:
        type: Integer
        min: 0
        sum+: Integer
```

Here, `sum+: Integer` is syntactic sugar for specifying that the sum of positive values must fit into a 32-bits integer – nicer than writing `max: 2147483647` or `max: 2**31-1` inside `sum+`. ■

3.3 The Python compilation

The specification for data validation expressed in YAML is compiled into Python code that can be processed by the ASP system CLINGO. The compilation injects data validation in the grounding process by introducing *constraint validators* of two kinds, namely *forward* and *implicit*, depending on the arity of the validated predicates and on how terms are passed to @-terms: for unary predicates, their unique terms are forwarded directly to the functions handling @-terms; for other predicates, instead, terms are grouped by functions with the same name of the validated predicate. Hence, for a predicate `pred` of arity 1, the (forward) constraint validator has the following form:

```
:- pred(X1), @valasp_validate_pred(X1) != 1.
```

Similarly, for a predicate `pred` of arity $n \geq 2$, the (implicit) constraint validator has the following form:

```
:- pred(X1,...,Xn), @valasp_validate_pred(pred(X1,...,Xn)) != 1.
```

In both cases, @-terms are associated with the Python function

```
def valasp_validate_pred(value):
    Pred(value)
    return 1
```

where `Pred` is a class whose name is obtained by capitalizing the first lowercase letter of `pred`, and whose constructor raises an exception if the provided data are invalid. In fact, class `Pred` is also an outcome of the compilation process, and materializes all validity conditions specified in the data validation specification in input.

In a nutshell, given a data validation specification (P, A, U) (represented in YAML and whose code blocks are written in Python), and an ASP program Π, the compilation produces a Python script with the following content:

1. The Python program P.
2. A Python class S for every $(s, D, H, b, c, a) \in U$ materializing all validity conditions: field declarations in D map to Python class annotations (and added as instance attributes on instance creation); field comparisons in H and the Python code block c are added to the `__post_init__` method (and executed after any instance creation); the Python code blocks b and a are respectively added to the class methods `before_grounding` and `after_grounding`.
3. Calls to any `before_grounding` method introduced in the previous steps.
4. Calls to CLINGO's API to ground the ASP program $\Pi \cup A \cup C$, where C is the set of constraint validators associated with U.
5. Calls to any `after_grounding` method introduced in the previous steps.

Example 3.7 (*Continuing Example 3.6*)
The YAML file to validate predicate `bday` of Example 3.1 is mapped to the following Python code:

```
import datetime

context = Context(wrap=[])

@context.valasp
class Date:
        year: Integer
        month: Integer
        day: Integer

        def __post_init__(self):
                datetime.datetime(self.year, self.month, self.day)

@context.valasp
class Bday:
        name: Alpha
        date: Date
```

The two Python classes, `Date` and `Bday`, are decorated with the decorator `@context.valasp`, which is in charge for interpreting the annotations used to declare fields: the constructor of (the decorated) `Date` class checks the presence of three integer arguments, namely `year`, `month` and `day`, and calls the `__post_init__` method to ensure that they form a valid date; the constructor of (the decorated) `Bday` class checks that the first argument is alphanumeric and the second argument is a valid instance of `Date`.

The YAML file to validate predicate `ordered_triple` of Example 3.2 is mapped to the following Python code:

```
@context.valasp
class Ordered_triple:
        first: Integer
        second: Integer
        third: Integer

        def __post_init__(self):
            if not self.first < self.second:
               raise ValueError("Expected first < second")
            if not self.second < self.third:
               raise ValueError("Expected second < third")
```

Note that the two `having` constraints are mapped to two conditional statements, and comprehensive messages are provided in case of violation.

Finally, the YAML file to validate predicate `income` of Example 3.3 is mapped to the following Python code:

```
@context.valasp
class Income:
        company: String
        amount: Integer

        def __post_init__(self):
            if self.amount < 0:
               raise ValueError(f"Should be >= 0, but received {self.amount}")
            if self.amount > 0:
                self.__class__.sum_positive_of_amount += self.amount

        @classmethod
        def before_grounding_init_positive_sum_amount(cls):
           cls.sum_positive_of_amount = 0

        @classmethod
        def after_grounding_check_positive_sum_amount(cls):
            if cls.sum_positive_of_amount > 2147483647:
               raise ValueError('sum of amount in income may exceed 2147483647')
```

Above we can observe that the `min` constraint is enforced in the first line of `__post_init__`, while the `sum+` constraint requires to initialize the class variable `sum_positive_of_amount` (in method `before_grounding_init_positive_sum_amount`), to update its value for each positive `amount` (in `__post_init__`), and a final check (in method `after_grounding_check_positive_sum_amount`). Note that integers in Python do not have fixed byte length. Therefore, there is no overflow in `sum_positive_of_amount`.

The decorated classes are then used for validation by means of the following code:

```
control = clingo.Control()
control.load("-")
try:
    context.valasp_run(control,
        on_validation_done=lambda: print("ALL VALID!\n=========="),
        on_model=lambda m: print(f"Answer: {m}\n=========="),
    )
except RuntimeError as e:
    raise ValueError(context.valasp_extract_error_message(e)) from None
```

Code similar to the above snippet is also produced by the translation of YAML files. As an alternative, validation can be specified directly in terms of the above Python classes, and the programmer can customize the invocation of VALASP. ■

4 Use cases and assessment

This section reports a few use cases on two encodings from ASP competitions (Gebser *et al.* 2020). Each use case focuses on the validation of parts of an encoding, showing how the proposed framework can identify invalid data. Note that tuning of the encoding is out of the scope of this paper. Moreover, the overhead introduced by data validation is empirically assessed. Finally, an application of VALASP for validating complex Python data structures is shown.

4.1 Video streaming – 7th ASP competition

Video streaming amounts to selecting an optimal set of video representations, in terms of resolution and bitrate, to satisfy user requirements. User requirements and solution are respectively encoded by `user(USERID, VIDEOTYPE, RESOLUTION, BANDWIDTH, MAXSAT, MAXBITRATE)` and `assign(USER_ID, VIDEO_TYPE, RESOLUTION, BITRATE, SAT)`. The overall satisfaction of users is maximized by the following weak constraint:

```
:~ assign(USER_ID,_,_,BITRATE,SAT_VALUE), user(USER_ID,_,_,_,BEST_SAT,_).
   [BEST_SAT-SAT_VALUE@1, USER_ID, assign]
```

According to the official description, available online at `http://aspcomp2015.dibris.unige.it/Video_Streaming.pdf`, instances of `user/6` can be validated with the following YAML specification:

```
user:
    userid:
        type: Integer
        min: 0
    videotype:
        type: String
        enum: [Documentary, Video, Cartoon, Sport]
    resolution:
        type: Integer
        enum: [224, 360, 720, 1080]
    bandwidth:
        type: Integer
        min: 0
    maxsat:
        type: Integer
        min: 0
    maxbitrate:
        type: Integer
        min: 150
        max: 8650
    valasp:
        after_init: |+
           if self.maxbitrate % 50 != 0: raise ValueError("unexpected bitrate")
```

According to the above specification, the arguments `userid`, `bandwidth` and `maxsat` are non-negative integers; `videotype` is a string among `Documentary`, `Video`, `Cartoon`, and `Sport`; argument `resolution` is an integer among 224, 360, 720, and 1080; and `maxbitrate` is an integer between 150 and 8650, and it is divisible by 50.

The official encoding and instances do not have errors, as expected. However, the encoding is quite fragile and relies on several assumptions on the input data and on ASP internals – ASP systems use 32-bits integers for everything but the cost of a solution. To show how dangerous such assumptions are, consider a decision problem where a partial solution and a target satisfaction are given. Accordingly, the weak constraint is replaced by the following constraint:

```
:- target(T), #sum{BEST_SAT-SAT_VALUE, USER_ID :
    assign(USER_ID,_,_,BITRATE,SAT_VALUE), user(USER_ID,_,_,_,BEST_SAT,_)} > T.
```

In this case, the execution of CLINGO on the instances of the competition may lead to the error message `"Value too large to be stored in data type: Integer overflow!"`, produced while simplifying the sum. However, whether the message is shown or not depends on the partial solution provided in input. In fact, if the overflow is only due to the `assign/5` instances in input, the subsequent simplification step cannot notice the problem and a wrong answer is produced. For example, if `BEST_SAT=1500000000` and the input contains two `assign/5` instances with `SAT_VALUE=1` and `SAT_VALUE=2`, then the grounder of CLINGO will simplify the aggregate by communicating to the solver that the minimum value is `(1500000000-1+1500000000-2) mod` 2^{31}`=852516349`, which is interpreted as 2^{31} `-852516349=-1294967299`; at this point, if the other values associated to the undefined instances of `assign/5` are not sufficient to overflow the sum, then the previous overflow is unnoticed. The following YAML specification can help to detect these overflows:

```
target:
    value:
      type: Integer
      min: 0
sum_element:
    value:
        type: Integer
        min: 0
        sum+: Integer
   userid: Integer
valasp:
    asp: |+
        sum_element(BEST_SAT-SAT_VALUE,UID) :-
            assign(UID,_,_,BITRATE,SAT_VALUE), user(UID,_,_,_,BEST_SAT,_).
```

4.2 Solitaire – 4th ASP competition

Solitaire represents a single-player game played on a 7×7 board where the 2×2 corners are omitted. We focus on the following rules defining the board:

```
range(1).
range(X+1) :- range(X), X < 7.
location(1,X) :- range(X), 3 <= X, X <= 5.
location(2,X) :- range(X), 3 <= X, X <= 5.
location(Y,X) :- range(Y), 3 <= Y, Y <= 5, range(X).
location(6,X) :- range(X), 3 <= X, X <= 5.
location(7,X) :- range(X), 3 <= X, X <= 5.
```

Those rules are interesting since an error in this point might be propagated all over the encoding. The YAML specification of `range` and `location` is the following:

```
range:
    value:
        type: Integer
        enum: [1, 2, 3, 4, 5, 6, 7]
location:
    x: range
    y: range
    valasp:
        after_grounding: |+
            pos = [1,2,6,7]
            if self.x.value in pos and self.y.value in pos:
                raise ValueError("Invalid position")
```

In particular, note that this example shows the usage of the `after_grounding` statement to check the valid positions after that all grounding instances of `location` have been created.

4.3 Qualitative spatial reasoning – 4th ASP competition

Qualitative spatial reasoning consists of deciding whether a set of spatial and temporal constraints is consistent with respect to a composition table. Membership in qualitative relations is encoded by 169 rules, similar to the following:

```
label(X,Z,rp) :- label(X,Y,rp), label(Y,Z,rp).
label(X,Z,req) | label(X,Z,rp) | label(X,Z,rpi) | label(X,Z,rd) |
label(X,Z,rdi)
| label(X,Z,rs) | label(X,Z,rsi) | label(X,Z,rf) | label(X,Z,rfi)
| label(X,Z,rm) | label(X,Z,rmi) | label(X,Z,ro) | label(X,Z,roi)
:- label(X,Y,rp), label(Y,Z,rpi).
```

The third argument of `label/3` is a qualitative relation. The following YAML specification can be used to validate such rules:

```
rel:
  value:
    type: Alpha
    enum: [req, rp, rpi, rd, rdi, ro, roi, rm, rmi, rs, rsi, rf, rfi]
node:
  value:
    type: Integer
    min: 0
    max: 49
label:
  x: node
  y: node
  l: rel
  valasp:
    having: [x < y]
```

The above example shows the usage of the `having` statement to compare the values of the two fields of the predicate `label`.

4.4 Knight's Tour – 3rd ASP competition

The knight's tour problem aims at finding a sequence of moves of a knight on a chessboard of size N such that the knight visits every square exactly once and comes back to the origin. We focus on a short excerpt of the encoding[2]:

```
size(8). givenmove(7,5,8,7). givenmove(1,7,3,6).

number(X) :- size(X).
number(X) :- number(Y), X=Y-1, X>0.
even :- size(N), number(X), N = X+X.
:- not even.
:- size(N), N < 6.
```

Those rules are particularly interesting from the point of view of the validation. First of all, the first line contains a test case, probably this was a test used by a programmer that was not commented out before publication. Note that atoms of the form givenmove/4 are part of the input, therefore the ones added in the encoding might lead to incorrect results. Moreover, remaining rules are used to check that the size of the chessboard must be an even number greater than 6, which we argue should not be part of the encoding.

The YAML specification of `size` is the following:

```
size:
  value:
    type: Integer
      min: 6
      max: 100
      count: 1

  valasp:
    after_init: |+
          if self.value %2 != 0:
            raise ValueError('Size must be an even number')
          self.__class__.value = self.value
```

In this case the validation will fail since there are two different atoms of the form size/1. In addition, the following YAML specification of `move` and `givenmove` can be used to validate knight's moves:

[2] The full version can be found at http://www.mat.unical.it/aspcomp2011/files/KnightTour/knight_tour-full_package.zip.

```
valasp:
  asp: |+
    __in_range(X1, givenmove(X1,Y1,X2,Y2)) :- givenmove(X1,Y1,X2,Y2).
    __in_range(Y1, givenmove(X1,Y1,X2,Y2)) :- givenmove(X1,Y1,X2,Y2).
    __in_range(X2, givenmove(X1,Y1,X2,Y2)) :- givenmove(X1,Y1,X2,Y2).
    __in_range(Y2, givenmove(X1,Y1,X2,Y2)) :- givenmove(X1,Y1,X2,Y2).

    __in_range(X1, move(X1,Y1,X2,Y2)) :- move(X1,Y1,X2,Y2).
    __in_range(Y1, move(X1,Y1,X2,Y2)) :- move(X1,Y1,X2,Y2).
    __in_range(X2, move(X1,Y1,X2,Y2)) :- move(X1,Y1,X2,Y2).
    __in_range(Y2, move(X1,Y1,X2,Y2)) :- move(X1,Y1,X2,Y2).

__in_range:
  x:
    type: Integer
    min: 1
    source: Any

valasp:
  before_grounding: |+
    cls.post_check = []

  after_init: |+
    self.__class__.post_check.append(self)

  after_grounding: |+
    for el in cls.post_check:
      if el.x > Size.value:
        raise ValueError(f'Value out of bound in {el.source}: {el.x}')
```

(Note that the above example uses f-strings; https://www.python.org/dev/peps/pep-0498/.)

4.5 Empirical assessment

The overhead introduced by VALASP to validate instances of the discussed problems was measured by running CLINGO with and without validation. The experiment was run on a 2.4 GHz Quad-Core Intel Core i5 with 16 GB of memory. VALASP was executed with the command-line option `--valid-only`, and CLINGO was executed with its Python interface; in both cases we disabled the computation of stable models since VALASP has no impact on the solving procedure. We remark here that the running time of VALASP includes grounding time. For each benchmark, we considered all available instances. Results are reported in Table 1.

Concerning video streaming, the average running time of CLINGO is 0.06 s, and the average running time of VALASP is 0.18 s. As for Solitaire, the average running time of CLINGO and VALASP is respectively 0.07 and 0.13 s. Concerning qualitative spatial reasoning, the average running time of CLINGO is 3.23 s, and the average running time of VALASP is 3.45 s. Finally, on knight's tour, the average running time of CLINGO and VALASP is respectively 0.27 and 0.50 s.

We can conclude that no significative overhead is eventually introduced by VALASP on these test cases.

Table 1. *Average running time (in seconds) of* CLINGO *and* VALASP *on tested benchmarks*

Benchmark	#	CLINGO	VALASP
Video streaming	43	0.06	0.18
Solitaire	27	0.07	0.13
Qualitative spatial reasoning	159	3.23	3.45
Knight tour	10	0.27	0.50

4.6 Application: VALASP *to validate Python data*

VALASP is not only a framework for the validation of ASP data, but also brings the declarative power of ASP to validate complex Python data. For example, consider a Python function F receiving in input a partially ordered set, that is, a binary relation being reflexive, symmetric, and transitive. The binary relation is stored in a Python data structure, for example a list of pairs or a sparse matrix. The Python function F works on the provided data under the assumption that it represents a partially ordered set. If input data is properly validated, the Python function should verify that the binary relation is actually reflexive, symmetric, and transitive. Usually, such a validation is achieved by implementing Python functions, with imperative and error-prone code. VALASP provides an alternative: the relation in input R can be mapped to ASP facts of the form `r(a,b)`, for all $(a, b) \in R$, for example with the help of a library like CLORM (https://github.com/potassco/clorm), and the following data validation specification can be used:

```
valasp:
    asp: |+
        element(X) :- r(X,Y).
        element(Y) :- r(X,Y).
        lost("reflexivity", X) :- element(X), not r(X,X).
        lost("symmetry", (X,Y)) :- r(X,Y), not r(Y,X).
        lost("transitivity", (X,Y,Z)) :- r(X,Y), r(Y,Z), not r(X,Z).

lost:
    property: String
    reason: Any
    valasp:
        after_init: |+
            raise ValueError(f"Lost {self.property} on {self.reason}")
```

If relation R is not a partially ordered set, then it misses at least one property among reflexivity, symmetry, and transitivity. Such a knowledge is encoded in the ASP program above, which eventually produces an instance of `lost/2`. According to the above data validation specification, VALASP will then execute the Python code block given in the `after_init`, thus raising an exception to inhibit the execution of function F on invalid data.

As another example of this kind, consider a Python function receiving in input an undirected graph and working under the assumption that the graph is connected. In order to validate such a precondition, the input graph can be mapped to the ASP predicates `vertex/1` and `edge/2`, and the following data validation specification can be used:

```
valasp:
    asp: |+
        connected(FIRST) :- FIRST = #min{X : node(X)}.
        connected(Y) :- connected(X), edge(X,Y).
        unconnected(X) :- node(X), not connected(X).

unconnected:
    node: Any
    valasp:
        after_init: |+
            raise ValueError(f"Unconnected node {self.node}")
```

If the input graph is not connected, an exception is raised, pointing to the unconnected node.

Empirical assessment. In order to evaluate the performance of VALASP to validate Python data we considered an implementation of the Prim's algorithm for computing a minimum spanning tree (Prim 1957). Indeed, a minimum spanning tree can be computed only for connected graphs, therefore we can use the validation presented above to check whether the input graph of the Python function is connected. We executed the experiment on instances of the problem Graceful Graphs submitted to the 7th ASP Competition, since they all contain connected graphs. Moreover, we associated each edge of the graph with a positive weight. For each instance, we executed the Python function with and without validation. In the first case, the average running time was 0.06 s, whereas in the second case it was 0.05 s.

5 Related work

The use of types in programming languages eases the representation of complex knowledge, favors the early detection of errors and provides an implicit documentation of source codes (Pierce 2002). For example, by stating that the arguments of predicate `bday` are of types `person_name` and `date`, there is no need to document the way these elements are represented, and any attempt to instantiate this predicate with different types is blocked. ASP-Core-2 (Calimeri *et al.* 2020), on the other hand, is untyped: there is no way to state that arguments of a predicate must be of a specific type, the language offers a very limited set of primitive types, and there is no idiomatic way to declare user-defined types. This work targets ASP-Core-2, the standardized language implemented by CLINGO (Gebser *et al.* 2018) and DLV2 (Adrian *et al.* 2018), aiming at providing the missing idioms to specify types and to validate data.

Types are not new in logic-based languages, and in particular order-sorted logic has been formalized as first-order logic with sorted terms, where sorts are ordered to build a hierarchy (Kaneiwa 2004). IDP3 (Cat *et al.* 2018) and SPARC (Marcopoulos and Zhang 2019) are two systems with languages close to ASP-Core-2 and supporting sorted terms. There are many differences between these systems and the framework proposed in this work. First of all, VALASP is designed to be smoothly integrated with ASP-Core-2 projects: the programmer is free to choose what to validate and what to leave unchecked, and the original encoding can still be used as it is in case validation is not required in the deployed software. Sorted terms are also used to bound object variables in rules, while this is not possible with VALASP because it only deals with the aspect of data validation.

The framework uses @-terms to perform data validation by means of Python functions that are called during the grounding process. In the literature, @-terms and non-Herbrand functions (Balduccini 2013) were used to enrich ASP with functionality that are otherwise not viable (if not in the Turing tarpit). External atoms in HEX (Eiter *et al.* 2018) extend the notion of externally interpreted function to externally interpreted relations, and can be also used to achieve some form of data validation (Redl 2017).

Intuitively, the constraints

```
:- pred(X1), @valasp_validate_pred(X1) != 1.
:- pred(X1,...,Xn), @valasp_validate_pred(pred(X1,...,Xn)) != 1.
```

can be replaced by the following HEX rule:

```
:- &valasp_validate_pred[pred]().
```

The implementation of the external atom `valasp_validate_pred` is similar to the implementation of the @-term `valasp_validate_pred`: it must call the constructor of the (decorated) class `Pred` produced by VALASP and return the empty relation; if the construction of `Pred` fails, VALASP raises an error and blocks the grounding of the program, and otherwise the empty relation returned by the external atom is such that the above constraint is satisfied. Hence, external atoms can be used as an alternative to @-terms for implementing the validation constraints defined in Section 3.3.

Finally, there are works in the literature that introduce data validation in Prolog systems (Kiel and Schader 1991) and that implement data validation for Constraint Logic Programming by means of Prolog systems (Hermenegildo *et al.* 2002; Puebla *et al.* 2000). The goal of those works is clearly related to this paper, but they differ on the way data validation is specified, on the target language and on the underlying implementation. Similarly, debugging techniques for ASP (Fandinno and Schulz 2019; Gebser *et al.* 2008; Oetsch *et al.* 2010; Dodaro *et al.* 2019) share the goal to identify errors, but with a different approach. VALASP aims at blocking data validation errors in a very early stage, at coding time and by implementing fail-fast techniques to point to the source of the problem. Debugging techniques instead are useful to localize the origin of unintended behavior, and usually require interaction with the programmer. If VALASP is properly used, a debugger is still a useful software in the tool belt of an ASP programmer, but on the other hand it is likely that the number of debugging sessions will be reduced. Moreover, VALASP is non-intrusive, since it does not require any change to the tested ASP program, differently from other recently-proposed techniques (Amendola *et al.* 2021; Lifschitz 2017).

6 Conclusion

ASP programmers do mistakes, there is no shame in this. VALASP aims at early detection of data validity errors, and promotes a fail-fast approach so that the origin of the problem can be quickly identified and fixed. The proposed approach follows the separation of concerns design principle: validation rules are specified in YAML with Python and ASP snippets, and are separated from the business logic represented in ASP encodings. Such a design is useful to smoothly introduce data validation in ASP, as validation rules can be specified externally without the need to deeply change the way programs are written. If after deployment data can be safely assumed valid, VALASP can be easily discharged because the original encoding stays unchanged. Moreover, VALASP opens the possibility to take advantage of ASP declarativity for validating complex Python data structures, bringing the expression of data validation specifications at a higher level of abstraction.

Finally, albeit VALASP has a tight integration with the state-of-the-art solver CLINGO, it can already be used in combination with other ASP solvers based on the ASP-Core-2 standardized language, for example, DLV. In particular, a user can:

- use VALASP for validation, and DLV for execution during the development phase; or
- use VALASP for validation and grounding, and DLV for stable model search (e.g. by using the option `--mode=wasp`) during the deployment phase; or
- use VALASP only during the development phase for early detection of bugs, and then use DLV without validation during the deployment phase.

Moreover, it is important to observe that VALASP is mainly based on @-terms, a feature that is implemented in CLINGO but still unavailable in DLV. This is currently the most difficult technical aspect to overcome in order to integrate VALASP in DLV.

Conflicts of interests

The authors declare none.

References

Adrian, W. T., Alviano, M., Calimeri, F., Cuteri, B., Dodaro, C., Faber, W., Fuscà, D., Leone, N., Manna, M., Perri, S., Ricca, F., Veltri, P. and Zangari, J. 2018. The ASP system DLV: Advancements and applications. *KI 32,* 2-3, 177–179.

Alviano, M. and Dodaro, C. 2020. Data validation for answer set programming (extended abstract). In *Proceedings of ICLP (Technical Communications)*. Electronic Proceedings in Theoretical Computer Science, EPTCS, vol. 325. Open Publishing Association, 93–95.

Alviano, M., Dodaro, C. and Zamayla, A. 2021. Data validation meets answer set programming. In *Practical Aspects of Declarative Languages - 23rd International Symposium, PADL 2021, Copenhagen, Denmark, January 18–19, 2021, Proceedings*, J. F. Morales and D. A. Orchard, Eds. Lecture Notes in Computer Science, vol. 12548. Springer, 90–106.

Alviano, M., Faber, W. and Leone, N. 2010. Disjunctive ASP with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming 10,* 4-6, 497–512.

Amendola, G., Berei, T. and Ricca, F. 2021. Testing in ASP: Revisited language and programming environment. In *JELIA*. Lecture Notes in Computer Science, vol. 12678. Springer, 362–376.

Balduccini, M. 2013. ASP with non-herbrand partial functions: A language and system for practical use. *Theory and Practice of Logic Programming 13,* 4-5, 547–561.

Baselice, S. and Bonatti, P. A. 2010. A decidable subclass of finitary programs. *Theory and Practice of Logic Programming 10,* 4-6, 481–496.

Calimeri, F., Cozza, S., Ianni, G. and Leone, N. 2011. Finitely recursive programs: Decidability and bottom-up computation. *AI Communications 24,* 4, 311–334.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F. and Schaub, T. 2020. ASP-Core-2 input language format. *Theory and Practice of Logic Programming 20,* 2, 294–309.

Cat, B. D., Bogaerts, B., Bruynooghe, M., Janssens, G. and Denecker, M. 2018. Predicate logic as a modeling language: The IDP system. In *Declarative Logic Programming: Theory, Systems, and Applications.* ACM/Morgan & Claypool, 279–323.

Dodaro, C., Alviano, M., Faber, W., Leone, N., Ricca, F. and Sirianni, M. 2011. The birth of a WASP: Preliminary report on a new ASP solver. In *Proceedings of CILC.* CEUR Workshop Proceedings, vol. 810. CEUR-WS.org, 99–113.

Dodaro, C., Gasteiger, P., Reale, K., Ricca, F. and Schekotihin, K. 2019. Debugging non-ground ASP programs: Technique and graphical tools. *Theory and Practice of Logic Programming 19,* 2, 290–316.

Eiter, T., Germano, S., Ianni, G., Kaminski, T., Redl, C., Schüller, P. and Weinzierl, A. 2018. The DLVHEX system. *KI 32,* 2-3, 187–189.

Eiter, T. and Simkus, M. 2009. Bidirectional answer set programs with function symbols. In *Proceedings of IJCAI*, 765–771.

Fandinno, J. and Schulz, C. 2019. Answering the "why" in answer set programming - A survey of explanation approaches. *Theory and Practice of Logic Programming 19,* 2, 114–203.

Gebser, M., Kaminski, R., Kaufmann, B., Lühne, P., Obermeier, P., Ostrowski, M., Romero, J., Schaub, T., Schellhorn, S. and Wanko, P. 2018. The potsdam answer set solving collection 5.0. *KI 32,* 2-3, 181–182.

Gebser, M., Maratea, M. and Ricca, F. 2020. The seventh answer set programming competition: Design and results. *Theory and Practice of Logic Programming 20,* 2, 176–204.

Gebser, M., Pührer, J., Schaub, T. and Tompits, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proceedings of AAAI.* AAAI Press, 448–453.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3/4, 365–386.

Hermenegildo, M. V., Puebla, G., Bueno, F. and López-García, P. 2002. Program debugging and validation using semantic approximations and partial specifications. In *Proceedings of ICALP.* LNCS, vol. 2380. Springer, 69–72.

Johnsson, D. B., Deogun, D. and Sawano, D. 2019. *Secure by Design.* Manning Publications.

Kaneiwa, K. 2004. Order-sorted logic programming with predicate hierarchy. *Artificial Intelligence 158,* 2, 155–188.

Kiel, R. and Schader, M. 1991. A tool for validating prolog programs. In *Classification, Data Analysis, and Knowledge Organization.* Springer, 183–188.

Ko, A. J. and Myers, B. A. 2003. Development and evaluation of a model of programming errors. In *Proceedings of HCC.* IEEE Computer Society, 7–14.

Ko, A. J. and Myers, B. A. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing 16,* 1-2, 41–84.

Leone, N., Allocca, C., Alviano, M., Calimeri, F., Civili, C., Costabile, R., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Cuteri, B., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P. and Zangari, J. 2019. Enhancing DLV for large-scale reasoning. In *Proceedings of LPNMR.* LNCS, vol. 11481. Springer, 312–325.

Lierler, Y. and Lifschitz, V. 2009. One more decidable class of finitely ground programs. In *Proceedings of ICLP.* LNCS, vol. 5649. Springer, 489–493.

Lifschitz, V. 2017. Achievements in answer set programming. *Theory and Practice of Logic Programming 17,* 5-6, 961–973.

Marcopoulos, E. and Zhang, Y. 2019. OnlineSPARC: A programming environment for answer set programming. *Theory and Practice of Logic Programming 19,* 2, 262–289.

Marek, V. W. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*. Springer Verlag, 375–398.

Natella, R., Winter, S., Cotroneo, D. and Suri, N. 2020. Analyzing the effects of bugs on software interfaces. *IEEE Transactions on Software Engineering 46,* 3, 280–301.

Niemelä, I. 1999. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25,* 3–4, 241–273.

Oetsch, J., Pührer, J. and Tompits, H. 2010. Catching the ouroboros: On debugging nonground answer-set programs. *Theory and Practice of Logic Programming 10,* 4-6, 513–529.

Padhye, R. and Sen, K. 2019. Efficient fail-fast dynamic subtype checking. In *Proceedings of VMIL@SPLASH*. ACM, 32–37.

Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press.

Prim, R. C. 1957. Shortest connection networks and some generalizations. *The Bell System Technical Journal 36,* 6, 1389–1401.

Puebla, G., Bueno, F. and Hermenegildo, M. V. 2000. A generic processor for program validation and debugging. In *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*. LNCS, vol. 1870. Springer, 63–107.

Redl, C. 2017. Extending answer set programs with interpreted functions as first-class citizens. In *Proceedings of PADL*. LNCS, vol. 10137. Springer, 68–85.

Vernon, V. 2016. *Domain-Driven Design Distilled*. Addison-Wesley.