Optimal Language Design is Hard

A Case Study in ECMAScript (JavaScript) Standardization

Philipp Riemer
Institut für Informatik
Leipzig University
Leipzig, Germany
pr83zyfu@studserv.uni-leipzig.de

Yury Nikulin
Department of Mathematics and Statistics
University of Turku
Turku, Finland
yurnik@utu.fi

Ashley Claymore
Bloomberg L.P.
London, UK
aclaymore@bloomberg.net

Mikhail Barash
Bergen Language Design Laboratory
University of Bergen
Bergen, Norway
mikhail.barash@uib.no

Abstract

In addition to requirements of a purely technical nature, the evolution of *widely adopted* programming languages is often governed by preferences of individual members—either persons or organizations—of the language maintenance team, who may associate issues with particular design aspects. For example, when adding a new primitive data type to a language, language designers may suggest alternative semantics for *equality to existing data types*, as well as explicitly specify issues they wish to avoid (such as inconsistent behavior among number-like types). The decision-making process can span over multiple years and can be highly unstructured, and in a dynamic and distributed language design team, the cumulated understanding of previously discussed design alternatives can thus be lost over time.

In this paper, we present a domain-specific language to specify a certain kind of language evolution proposals—where the design space can be presented as a collection of interconnected individual design points. With each design point, one can associate a set of issues it could raise that should be avoided. From a DSL specification, an interactive web-based tool is generated that allows exploring the design space of a proposal.

Further assigning weights to issues, we formulate an optimization problem where the goal is to select alternatives for individual design points to minimize the total weight of occurring issues. We prove that this optimization problem is NP-hard. We reduce this problem to an integer linear programming problem and incorporate a solver into our interactive tool. We demonstrate the feasibility of our approach by using our DSL to specify the ECMAScript proposal *Records & Tuples*, and demonstrate that any design choice—as described in the proposal—will necessarily raise issues.

CCS Concepts: **• Software and its engineering → Domain specific languages**; **General programming languages**; **Software evolution**; **• Computing methodologies → Modeling and simulation**; **• Applied computing → Multi-criterion optimization and decision-making**; **Decision analysis**.

Keywords: Language evolution, widely adopted languages, domain-specific languages, decision making, linear programming.

ACM Reference Format:
Philipp Riemer, Yury Nikulin, Ashley Claymore, and Mikhail Barash. 2025. Optimal Language Design is Hard: A Case Study in ECMAScript (JavaScript) Standardization. In *Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering (SLE '25), June 12–13, 2025, Koblenz, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3732771.3742715

1 Introduction

It is a common consensus that the engineering of software languages—either domain-specific or general-purpose—with a smaller user community differs significantly from the engineering of *widely adopted* software languages. This especially manifests when considering *evolution* and *maintenance*, where the widely adopted language is used by potentially millions of developers and every change to the language specification is bound to have a significant impact.

While some widely adopted languages are designed by small teams within a business entity, the evolution of other languages is an effort led by several players [1]. For instance, the evolution of Swift[1], Go[2], and Kotlin[3] is mainly driven

[1] https://github.com/swiftlang/swift-evolution/blob/main/process.md
[2] https://github.com/golang/proposal
[3] https://kotlinfoundation.org/language-committee-guidelines/

This work is licensed under a Creative Commons Attribution 4.0 International License.

SLE '25, Koblenz, Germany

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1884-7/2025/06
https://doi.org/10.1145/3732771.3742715

by groups of employees within their respective companies, while the evolution processes of ECMAScript, C++, Python, or HTML are highly formalized and follow an openly described process [2–5].

Large language design teams attempt to overcome the "design by committee" [6] approach, and tend to form *ad hoc* working groups (sometimes called "champion groups") that work on the design of a particular language feature. The rest of the language design team can then focus on reviewing and approving the design—rather than on actively participating in the design process itself. A thorough language design assumes a multi-faceted exploration of a feature's properties, and in many cases, the design space of a proposal becomes fine-grained, and an input from multiple stakeholders needs to be gathered. Thus the design may become contentious. This oftentimes comes from requirements whose nature is not purely technical (such as a lack of resources to implement a particular subfeature, lack of community support, or the presence of a competitive design of the feature).

When exploring a design space of a feature, members of the language design team can identify various issues with particular aspects of the design (for instance, memory-safety concerns, non-orthogonality, and so on). Each member of the language design team thus cumulates their own understanding of the design space. These members' perceptions then form the backbone of the discussions at the dedicated working group's meetings during which decisions on the feature design are made.

For languages whose standardization efforts are guided by a formally established standardization body, the timespan during which a feature is being designed is usually measured in years. This can be particularly challenging for dynamic teams, where members of the design team leave and join on a frequent basis. Recently joined language designers tend to commonly offer seemingly "straightforward" solutions that have already been discussed and rejected by the working group. Similarly, in geographically distributed language design teams—where some members are only able to attend a portion of the meetings—the working group may find a solution that satisfies the participants of a particular meeting, only to then have that solution raise another issue in a subsequent meeting where the participants would be different. The motivation of our work is to provide language designers with a way to explore their draft solution "offline" (i.e., outside of a formal meeting), as well as explore issues raised by others.

In this paper, we consider one of the proposals currently[4] designed by the Ecma International Technical Committee 39 (TC39), the group responsible for the design of the programming language ECMAScript (informally known as JavaScript). The proposal *Records & Tuples* [7] introduces two new primitive value types into the language: *record* (deeply immutable object-like structure, with the syntax `#{x:0,y:0}`) and *tuple* (deeply immutable array-like structure, with the syntax `#[1,2,3]`). Incorporating these two new primitive datatypes into the language requires establishing several properties relating these datatypes to already existing ones. For example, it is necessary to define the semantics of equality, especially in the corner cases[5], or whether objects—which are inherently mutable in ECMAScript—can be members of records and tuples, and so on. The design space of this proposal is non-trivial; this claim is supported by the fact that the GitHub repository that specifies the proposal has over 190 issues over the period of five years, of which 15 are labelled as "undecided point"[6] as of March 2025. To assist themselves in the decision-making around the design of this proposal, members of the TC39 committee developed an interactive tool [8] that allows exploring various design aspects of the proposal. In this tool, the design space is presented as a collection of individual *design points*, and a member of the proposal design team can make a separate decision about each of them. With each design point, (other) members of the proposal design team may associate a set of *issues*. These issues become "raised" (i.e., reported to the language designer) when a particular decision has been made about a particular design point. Thus, a member of the design team has a detailed overview of others' concerns about a particular design choice, and discussions on the proposal tend to proceed in a more structured manner.

The interactive tool [8] was implemented in an *ad hoc* manner specifically for the *Records & Tuples* proposal. One can observe that similar tools can be beneficial for exploring design spaces for other proposals. In this paper, we define a *domain-specific language to specify individual design points of a design space, as well as issues that are associated with the design points*. Based on a specification in the DSL, a web-based interactive tool is generated. This tool allows a language designer to explore the design space by making various decisions regarding the individual design points, and analyzing the issues raised by the design choices. The functionality of interactive tools generated from DSL specifications replicates that of the *ad hoc* tool for the *Records & Tuples* proposal used by TC39.

One can assume that when working with the interactive tool, a language designer will naturally try to make design decisions in a way that will minimize the amount of issues which are raised by the design. In the paper, we formalize this intuition as a mathematical optimization problem. The goal is to make decisions regarding individual design points in a way that would mimimize the amount of the raised issues. In a more general formulation, issues can be assigned weights; then the goal is to minimize the total weight of

[4]At the time of writing (March 2025).

[5]https://github.com/tc39/proposal-record-tuple/issues/65

[6]https://github.com/tc39/proposal-record-tuple/issues?q=label%3A%22undecided%20point%22

all issues raised by a design choice. We prove that this optimization problem in NP-hard, meaning that there are no known algorithms to solve the problem in polynomial time. We reduce our problem to the well-known Binary Linear Programming [9] problem, for which practical solutions exist [10]. These solutions are supported by standard solvers, and we use the SCIP toolset [11, 12]. Thus, *from a specification in our DSL, we generate an interactive tool that supports finding the optimal design of a feature, with respect to the design points specified by the language design team.*

We demonstrate the feasibility of our approach by specifying the design space of the ECMAScript *Records & Tuples* proposal in our DSL, and establish that any optimal design of this proposal will have at least three issues raised.

2 SpecAlt: a DSL for Specifying Design Alternatives

We introduce a domain-specific language (DSL) for specifying design spaces, give examples of how fragments of the ECMAScript *Records & Tuples* proposal can be expressed in the DSL, and formalize the notion of a "design template".

2.1 Domain-Specific Language for Design Alternatives

The two central concepts in the language are *tweakables* and *issues*. Each tweakable represents an individual design point within a design space. This design point has a *label* (defined, for example, as an expression in the language being designed, or as a description in prose), and a set of *values*, which represent the possible choices for this design point. With each value—that is, with each choice for a tweakable—one can associate a set of issues that will be raised in case that particular choice becomes preferred ("selected") by a language designer during the interactive design process. Syntactically, a tweakable is specified as follows:

```
1  tweakable <identifier> {
2    expression "<string-literal>"
3    default value value_1
4    value value_2 {
5      raise <issue-identifier> [when
           <condition-identifier>]
6      // more `raise` statements
7    }
8    // more `value` blocks
9    value value_N {
10     raise <issue-identifier> [when
           <condition-identifier>]
11     // more `raise` statements
12   }
13   [disabled {
14     message "<string-literal>" when
           <condition-identifier>
15     // more `message` statements
16   }]
17 }
```

Values can be represented either by string literals or identifiers. One can specify several values for a tweakable, and one of those values can be prepended by the modifier `default`, representing the fact that that particular value will be preselected for the tweakable during the interactive design process.

Within a `value`-block, multiple `raise`-statements can be specified. Each such statement specifies the name of the issue that will be raised in case the value mentioned in the enclosing `value`-block gets selected for the tweakable by a language designer. Additionally, a `raise`-statement can specify a Boolean expression—a *condition*—which has to evaluate to true for the issue to be actually raised.

A condition specification is a Boolean expression alongside with a name for it[7]:

```
1  condition <ident> holds when <bool-expr>
```

Boolean expressions are defined inductively on their structure in a standard manner:

- if t is a tweakable and v is a valid value for t, then t `is` v and t `is not` v are Boolean expressions;
- if e is a Boolean expression, then `(` e `)` and `not` e are Boolean expressions;
- if e_1 and e_2 are Boolean expressions, then e_1 `and` e_2 and e_1 `or` e_2 are Boolean expressions.

An issue specification mentions its name, a short summary, and a complete description.

```
1  issue <identifier> {
2      summary "<string-literal>"
3      description "<string-literal>"
4  }
```

Finally, one can specify when a tweakable should become *disabled*, i.e., when it should not be possible to assign a value to it. The conditions for disabling a tweakable are specified within an optional `disabled`-block; each condition has a corresponding explanatory message which will appear in the interactive tool. If several conditions are specified, then the tweakable becomes disabled if at least one of those conditions evaluates to true. For an issue to be enabled, all of the conditions must evaluate to false.

[7] To improve the ergonomics of using the DSL, the conditions in the `when`-clauses of the `raise`-statements can be inlined, without the need to define a separate `condition` declaration.

2.2 Example: Designing the Semantics of Equality Testing for a New Primitive Datatype in ECMAScript

We showcase now a specification of the two tweakables in the ECMAScript *Records & Tuples* proposal using our DSL[8].

Example 2.1. Consider how the semantics of equality can be defined for tuples with a single element whose value is `NaN`[9]. In JavaScript, equality testing can be done, for example, using the *strict equality* operator ("==="[10]). The outcome of the strict equality test can either be "true" or "false"—and it is a task of a language designer to determine what the outcome should be for the case of tuples with a single `NaN` element. This task establishes an individual design point in the overall design space for the tuples primitive data type.

If a language designer determines the outcome of this equality testing to be "false", this may lead to an inconsistency of how the equality is defined. One of the champions of the *Records & Tuples* proposal has summarized this consistency change as follows [8]: *"Currently the only value not equal to itself is NaN, and this can be used as a reliable check for NaN. If any record or tuple containing a NaN within its tree is also not equal to itself, then there would be an infinite number of values not equal to themselves."* Thus, based on this observation, it is reasonable to associate an issue which should be raised in case a language designer selects value "false" as an outcome for strict equality testing for tuples with a single `NaN` value.

In our DSL, this design point can be represented as a tweakable as follows:

```
1 tweakable tupleNaNAreTripleEqual {
2     expression "#[NaN] === #[NaN]"
3     default value true {}
4     value false {
5        raise unequalTupleNaN
6     }
7 }
```

The label of the tweakable is represented by a pseudo-quoted JavaScript expression (`"#[NaN] === #[NaN]"`), and the tweakable has two possible values—`true` and `false`[11]. When an interactive design tool will be generated from the DSL code, this tweakable will be pre-assigned the default value `true`. The user of the tool (i.e., another language designer) will be able to select between the values `true` and `false` for this tweakable. If value `false` is selected, the issue `unequalTupleNaN` will be raised. This issue can be defined in our DSL as follows:

```
1 issue unequalTupleNaN {
2     summary "consistency change"
3     description "a longer description"
4 }
```

Another approach for equality testing in JavaScript is using the *same-value equality* comparison (i.e., `Object.is` method[12]). As is the case for all equality testing approaches, the outcome of the same-value equality test can be either "true" or "false". As observed by a proposal champion [8], *"if both 'Object.is(NaN, NaN)' and '#[NaN] === #[NaN]" are true, there does not appear to be a reason for Object.is(#[NaN], #[NaN]) to not be true."* This observation necessitates raising an issue if a language designer determines the outcome of the same-value equality as "false" *and—at the same time*—the outcome of the strict equality as "true". This can be specified as the following tweakable in our DSL:

```
1 tweakable
     tupleWithNanObjectIsTupleWithNaN {
2   expression "Object.is(#[NaN], #[NaN])"
3   default value true {}
4   value false {
5     raise nanNotIsNaN
6     when tupleNaNAreTripleEqual is true
7   }
8 }
```

The `raise`-statement will raise the issue `nanNotIsNaN` on the condition that the value of the tweakable `tupleNaN-AreTripleEqual` (defined above) is `true`.

The issue `nanNotIsNaN` can be specified as follows:

```
1 issue naNNotIsNaN {
2    summary "Object.is NaN semantics"
3    description "a longer description"
4 }
```

□

We have now showcased our DSL by specifying two individual—yet inter-dependent—design points of a non-trivial design space. In the next example, we demonstrate the use of `disabled`-statements in tweakable specifications.

Example 2.2. The primitive datatype *tuple* introduced in the *Records & Tuples* proposal is immutable. In particular, this means that tuples cannot hold JavaScript objects—for objects are inherently mutable and are stored by reference rather than copied by value—without breaking the tuple's deep immutability guarantee.

As a possible solution to this problem, the *Records & Tuple* proposal suggests introducing the concept of *boxes*, using which one can encapsulate a mutable object in an immutable structure: a box should prevent accidental modifications to

[8]A complete specification of the *Records & Tuples* proposal in our DSL is discussed in Section 5.
[9]https://github.com/tc39/proposal-record-tuple/issues/65
[10]https://262.ecma-international.org/#sec-isstrictlyequal
[11]Note that in our DSL, these values are identifiers and cannot be evaluated per se.
[12]https://262.ecma-international.org/#sec-object.is

the encapsulated object. The proposal also discusses an alternative approach to store objects in tuples without introducing the box type (namely, by using Symbols as WeakMap keys)[13].

Since there are two alternative approaches mentioned in the proposal, there is a branching point in the design space: some of the design points will only be relevant if the box type is actually defined in the language. Thus, when exploring the design space, a language designer can "switch on and off" the presence of the box type in the language. In our DSL, the following tweakable can be defined to specify this:

```
1  tweakable typeofBoxConstructor {
2     expression "typeof Box"
3     default value "undefined" { /*...*/ }
4     value "function" { /*...*/ }
5  }
```

Assuming the Box type is defined, a language designer can select values of other relevant tweakables. An example of such a tweakable is testing the type of the tuple with a single element that is an empty object.

```
1  tweakable typeOfTupleWithBox {
2     expression "typeof #[Box({})]"
3     default value "tuple" { /*...*/ }
4     value "object" { /*...*/ }
5     disabled {
6        message "Box is not defined"
7        when typeofBoxConstructor
8             is "undefined"
9     }
10 }
```

The disabled-statement here models the requirement that the tweakable typeOfTupleWithBox should be disabled when the Box type is not defined (i.e., in cases when a language designer sets the value of the tweakable typeofBoxConstructor to "undefined"). □

2.3 Formalizing Design Choices

We are now ready to formally describe tweakables and issues; this will enable further analyses of design specifications in our DSL.

Throughout the rest of the paper, we denote by L the language currently being designed. Let E be a set of *expressions* (represented as strings), and let V be a set of *values* (represented as either strings or identifiers).

Definition 2.3. Let f be a feature being designed for a language L. A *design template* of f is a triple $D_f = \langle C, A, T \rangle$, where:

- C is a finite set of symbols representing *issues*;
- A is a finite set of *issue activations* of the form $b \Rightarrow c$, where $c \in C$, and b is a Boolean proposition over terms $t = v$, $t \neq v$ and $\top$, with $t \in T$, $v \in V$, and $\top$ representing the logical truth;
- T is a finite set of *tweakables* of the form

$$t ::= \underline{\texttt{select}}\ e\ \underline{\texttt{of}} \\ \underline{\texttt{case}}\ v_1 : A_1 \\ \dots \\ \underline{\texttt{case}}\ v_n : A_k,$$

  where $e \in E$ is an expression, $v_1, \dots, v_n \in V$ are possible values of tweakable t, and sets $A_1 \subseteq A, \dots, A_k \subseteq A$ represent issues raised when the expression e has value $v_1, \dots, v_k$, respectively.

Intuitively, the machinery of a design template can be understood as follows: after a value for each tweakable has been selected, one identifies which issues associated with this value for the tweakable get raised ("activated") by analyzing Boolean conditions—informally "activation conditions" or "triggers"—in the issue activations.

Example 2.4. The DSL specification in Example 2.1 can be formally represented as a design template $D = \langle C, A, T \rangle$, where the set of issues is $C = \{c_1, c_2\}$, the set of tweakables is $T = \{t_1, t_2\}$, the set of issue activations is $A = \{\top \Rightarrow c_1, (t_1 = \text{true}) \Rightarrow c_2\}$, and tweakables t_1 and t_2 are defined as follows:

$$t_1 ::= \underline{\texttt{select}}\ \texttt{"\#[NaN] === \#[NaN]"}\ \underline{\texttt{of}} \\ \underline{\texttt{case}}\ \text{true} : \varnothing \\ \underline{\texttt{case}}\ \text{false} : \{\top \Rightarrow c_1\}$$

$$t_2 ::= \underline{\texttt{select}}\ \texttt{"Object.is(\#[NaN], \#[NaN])"}\ \underline{\texttt{of}} \\ \underline{\texttt{case}}\ \text{true} : \varnothing \\ \underline{\texttt{case}}\ \text{false} : \{(t_1 = \text{true}) \Rightarrow c_2\}$$

Note that $\top$ is used in the issue activation $\top \Rightarrow c_1$ to represent the unconditional raise of the issue. □

Note that Definition 2.3 does not explicitly formalize disabled tweakables. Indeed, let $d_1, \dots, d_n$ be conditions for disabling a tweakable t. Consider an activation $b \Rightarrow c$ corresponding to the tweakable t. The conditions d_i can now be incorporated directly into the activation as follows: $(b \wedge \neg d_1 \wedge \dots \wedge \neg d_n) \Rightarrow c$.

Given a design template, a language designer can assign a value to each tweakable, thus completely—to a known extent—specifying the design space of a feature. This is formalized in the following definition.

Definition 2.5. Let $D_f = \langle C, A, T \rangle$ be a design template of a feature f in a language L. A *design choice* $S : T \rightarrow V$ is an assignment of values $v_1, \dots, v_m$ to tweakables $t_1, \dots, t_n$, where each value $v_j \in V$ is one of the possible values for tweakable $t_i \in T$. Thus, design choice S induces an interpretation I_S

[13] https://tc39.es/proposal-record-tuple/tutorial/#keeping-track-of-objects-in-record--tuple

over the Boolean propositions present in the issue activations A, such that:

$$S(t_i) = v_j \quad \Leftrightarrow \quad I_S(t_i = v_j) = 1 \quad \Leftrightarrow \quad I_S(t_i \neq v_j) = 0.$$

The set of all raised issues for a given design choice can be defined as follows.

Definition 2.6. Let S be a design choice for a design template $D = \langle C, A, T \rangle$. For each tweakable $t \in T$, let $S(t) = v$ for some valid value v. Let A_v be the set of issue activations corresponding to v in the specification of t:

$$\begin{aligned} t ::= \underline{\texttt{select}}\ e\ \underline{\texttt{of}} \\ \dots \\ \underline{\texttt{case}}\ v : A_v \end{aligned}$$

Consider issue activations of the form $(b \Rightarrow c) \in A_v$, for which the interpretation of the Boolean proposition b is $I_S(b) = 1$. Denote by Ψ_S the set of *all issues raised by S*, comprised of issues $c \in C$ in such activations.

We can demonstrate these definitions on an example.

Example 2.7. For the design template in Example 2.4, a possible design choice is $S(t_1) = \text{true}$, $S(t_2) = \text{false}$. For the value false of tweakable t_2, the set of issue activations only has one element $(t_1 = \text{true}) \Rightarrow c_2$, and the interpretation of the Boolean proposition is $I(t_1 = \text{true}) = 1$ since $S(t_1) = \text{true}$. Thus, the set of all raised issues for the design choice S is $\Psi_S = \{c_2\}$. □

3 Finding Design Choices with Least Issues

Consider the set of all raised issues for a given design choice. One can argue that a reasonable expectation from a language designer is that they will strive to select the values of tweakables in a way that would lead to the least possible amount of issues. This intuition can be formalized as an optimization problem.

3.1 Spec-Dec-Opt: Optimization Problem

As a straightforward quantitative measure—a "score"—of a design choice S, one could use the *amount of raised issues* in the set Ψ_S. In realistic settings, however, some of the issues will naturally be perceived as more severe than others. This is accounted for in the following definition.

Definition 3.1. Let S be a design choice of a design template $D = \langle C, A, T \rangle$. Denote by $w : C \to \mathbb{R}$ an ascription of severities to issues. Define *preferability* $\sigma_w(S)$ of the design choice S as sum of individual issues' severities, preceded by a negative factor:

$$\sigma_w(S) = -\sum_{c_i \in \Psi_S} w(c_i).$$

In case when the severities of all issues are set to $+1$, the preferability $\sigma_w(S)$ of a design choice S represents the amount of the raised issues. Negative severities can be used to indicate a positive effect on a design choice; this covers situations where designers want to encourage—but not require—certain design attributes.

Based on the severities of the issues in a design choice, we can classify issues as follows.

Definition 3.2. Let $D = \langle C, A, T \rangle$ be a design template and w be a severity ascription. An issue $c \in C$ is called:

- *blocking*, if $w(c) = +\infty$ (this issue alone invalidates the design choice in its entirety and the design choice preferability is thus $-\infty$);
- *negligible*, if $w(c) = 0$;
- *enabling*, if $w(c) = -\infty$ (this issue alone validates the design choice in its entirety and the design choice preferability is considered to be $+\infty$).

We are now ready to formulate a mathematical optimization problem of maximizing the preferability of a design choice.

> Spec-Dec-Opt: **Given a design template D and severity ascription w, find a design choice S that maximizes preferability $\sigma_w(S)$.**

We state below the complexity of this problem.

Theorem 3.3. Spec-Dec-Opt *is* **NP**-*hard.*

Proof. A polynomial reduction of the satisfiability problem SAT to Spec-Dec-Opt (see Appendix A). □

3.2 Practical Solution for Spec-Dec-Opt

Theorem 3.3 establishes that there are no known algorithms to solve our problem Spec-Dec-Opt in polynomial time. Below we present a practical approach to solve Spec-Dec-Opt by reducing it to a linear programming problem with well-known solutions.

> **Binary Linear Programming (BLP)** [9]: Given a constraint matrix $M \in \mathbb{R}^{m \times n}$, a right-hand side vector $\boldsymbol{d} \in \mathbb{R}^m$, and a coefficient vector $\boldsymbol{p} \in \mathbb{R}^n$, find a decision vector $\boldsymbol{x} \in \{0, 1\}^n$ which minimizes the linear objective function $\boldsymbol{p}^T \boldsymbol{x}$ subject to the set of linear constraints $M\boldsymbol{x} \leq \boldsymbol{d}$.

Reducing the Spec-Dec-Opt problem to BLP means that we can transform an instance of Spec-Dec-Opt into an instance of BLP in a such a way that the process of choosing a fitting design choice (in terms of Spec-Dec-Opt) is equivalent to the process of finding a decision vector $\boldsymbol{x}$ (in terms of BLP)[14]. We explain below how such a reduction can be defined.

[14] Note that Spec-Dec-Opt is a maximization problem, and BLP is a minimization problem.

The construction will move along the structure of the output, which represents the matrix M and the vectors $\boldsymbol{x}$ and $\boldsymbol{d}$, and is shown in Figure 1*(left)* schematically and in Figure 1*(right)* in detail. The values in matrix M can be grouped into four blocks: `T2L`, `N1`, `N2`, `C2L`, vector $\boldsymbol{d}$ can be grouped into two blocks `T2R` and `C2R`, and the components of the unknown decision vector $\boldsymbol{x}$ can be grouped into three blocks `T1`, `C1`, and `Z`. We shall explain how each of these blocks is constructed; since some of the blocks are dependent on each other, the order of their construction is significant—and this is the order which the presentation follows.

In the construction below, consider a design template $D = \langle C, A, T \rangle$ and a severity ascription $w : C \longrightarrow \mathbb{R}$.

`T1`*: "base" variables.* The components of the block `T1` of vector $\boldsymbol{x}$ encode the information about the selected values of the tweakables. Denote by $S_{\boldsymbol{x}}$ the design choice that $\boldsymbol{x}$ represents. For every $t \in T$ with cases $v_1 : A_1, \ldots, v_k : A_k$, we introduce Boolean variables $x_{t,v_1}, \ldots, x_{t,v_k} \in \{0, 1\}$, for which selecting value v_j for tweakable t implies that the value of the variable x_{t,v_j} is 1, and vice versa:

$$S_{\boldsymbol{x}}(t) = v_j \quad \Leftrightarrow \quad x_{t,v_j} = 1$$

`T2L`*,* `N1`*,* `N2`*,* `T2R`*: assignment validity.* For a design choice to be valid, one must ensure that only one value is selected for every tweakable, that is, only one of the variables x_{t,v_j} has value 1, and the rest have value 0:

$$\sum_{1 \leq j \leq k} x_{t,v_j} = 1$$

Since the formulation of the BLP problem only allows for inequalities, we rewrite the equality above as follows:

$$\sum_{1 \leq j \leq k} x_{t,v_j} \leq 1 \tag{1}$$

$$\sum_{1 \leq j \leq k} -x_{t,v_j} \leq -1 \tag{2}$$

The left-hand sides of these inequalities make up the block `T2L` in the matrix M, and the right-hand sides form the block `T2R` of vector $\boldsymbol{d}$.

The block `T2L` is a matrix with rectangular sub-blocks across its diagonal, as can be seen in detail in Figure 1*(right)*. Each such sub-block corresponds to a tweakable; the height of each sub-block is always 2 (to represent the values 1 and −1 in the right-hand side parts of the inequalities (1)–(2)), and the width of a sub-block matches the number of possible values for the corresponding tweakable.

Again, to represent the values 1 and −1 occurring in the inequalities (1)–(2), the block `T2R` of vector $\boldsymbol{d}$ consists of alternating values 1 and −1.

The remaining variables in $\boldsymbol{x}$ do not appear in the inequalities (1)–(2); this is represented by the blocks `N1` and `N2`, both of which are matrices with only zero values.

`C1`*: "raise" variables.* The process of minimizing the objective function $\boldsymbol{p}^T\boldsymbol{x}$ in the BLP setting must be equivalent to the process of maximizing the preferability of the design choice in the Spec-Dec-Opt setting. The optimal value is dependent on the exact set of issues raised by the design choice; this fact is encoded in the block `C1` of vector $\boldsymbol{x}$. This block is comprised of Boolean "raise" variables $r_1, \ldots r_{|C|} \in \{0, 1\}$, such that the value of a "raise" variable r_i is 1 if and only if the issue c_i is raised by the design choice $S_{\boldsymbol{x}}$:

$$c_i \in \Psi_{S_{\boldsymbol{x}}} \quad \Leftrightarrow \quad r_i = 1 \tag{3}$$

This encodes the design choice and the information necessary to determine its preferability.

Example 3.4. The specification from Example 2.4 can be represented as follows in the BLP setting:

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & \cdots \\ -1 & -1 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 1 & 1 & 0 & 0 & \cdots \\ 0 & 0 & -1 & -1 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} x_{t_1,v_{\text{true}}} \\ x_{t_1,v_{\text{false}}} \\ x_{t_2,v_{\text{true}}} \\ x_{t_2,v_{\text{false}}} \\ r_1 \\ r_2 \\ \vdots \end{pmatrix} \leq \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \\ \vdots \end{pmatrix}$$

Note that blocks `N2`, `C2L`, `Z`, and `C2R` have been so far omitted, and will be given later in Example 3.5. □

`C2L`*,* `Z`*,* `C2R`*: raising of issues.* These blocks specify the machinery of equation (3).

In the simplest case, issue c_i is raised *unconditionally*[15] when some tweakable t has value v. In terms of (3), this means that $r_i = 1 = x_{t,v}$, where $x_{t,v}$ is a "base" variable introduced in the block `T1`. This equality can be represented by two linear inequalities:

$$-x_{t,v} + r_i \leq 0$$
$$x_{t,v} - r_i \leq 0$$

The corresponding entries in the block `C2R` will be thus 0, and the corresponding vectors in the block `C2L` will be $(\cdots, -1, \cdots; \cdots, 1, \cdots)^T$ and $(\cdots, 1, \cdots; \cdots, -1, \cdots)^T$, respectively.

Similarly, when issue c_i is raised unconditionally when $t \neq v$, we can constrain variables r_i and $x_{t,v}$ to be unequal:

$$-x_{t,v} - r_i \leq -1$$
$$x_{t,v} + r_i \leq 1$$

The non-trivial case is when an issue is raised *conditionally*. A given issue may be raised in various tweakables under various conditions and their machinery needs to expressed as a compound Boolean expression, which will in its turn be encoded as linear constraints in the blocks `C2L`, `Z`, and `C2R`.

[15] In the *Records & Tuples* proposal, it is common for an issue to be only mentioned in a single `raise`-statement corresponding to a single value of a single tweakable, without any additional conditions for its activation.

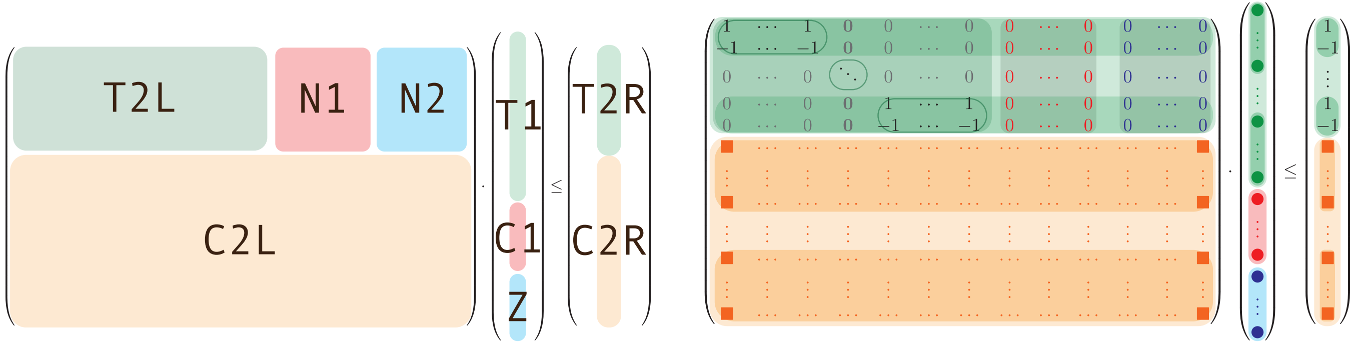

Figure 1. *(left)* Blocks in the matrix M and vectors $\boldsymbol{x}$ and $\boldsymbol{d}$ in the instance of the BLP problem, constructed from an instance of the Spec-Dec-Opt problem. *(right)* Elements of the constructed blocks. Square placeholders represent integer values from the set $\{-1, 0, 1\}$, and circle placeholders represent the decision vector's elements from the set $\{0, 1\}$.

For each tweakable $t \in T$ with cases $v_1 : A_1, \dots, v_k : A_k$, and issue $c \in C$, we define a set

$$\Pi(t, c) = \{(v_i, b_i) \mid (b_i \Rightarrow c) \in A_i, \quad 1 \le i \le k\}$$

that reflects the correspondence of a tweakable's value and the Boolean proposition in the respective issue activation. For an issue $c \in C$, we can now define a Boolean proposition ψ_c that will be true if and only if the issue c is raised for some value v of some tweakable $t \in T$:

$$\psi_c = \bigvee_{\substack{t \in T \\ (v,b) \in \Pi(t,c)}} ((t = v) \wedge b).$$

Note that this equation is defined in terms of tweakables and their values and issue activations. To incorporate such propositions into the matrix M and vectors $\boldsymbol{x}$ and $\boldsymbol{d}$, each ψ_c must be expressed in terms of "base" variables $x_{t,v}$. We denote this by $\psi_c(\boldsymbol{x})$, and—reformulating equation (3)—require that

$$\psi_{c_i}(\boldsymbol{x}) = 1 \quad \Leftrightarrow \quad r_i = 1$$

We explain now how a Boolean expression of the form ψ_c can be represented using linear constraints. In addition to "base" variables $x_{t,v}$ and "raise" variables r_i, it becomes necessary to introduce additional variables in the decision vector $\boldsymbol{x}$; this will be done in the block Z.

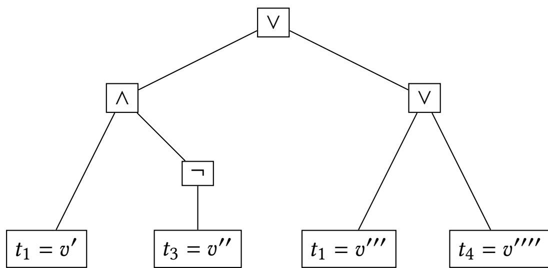

Figure 2. Example of the structure of the formula $\psi = (t_1 = v' \wedge \neg t_3 = v'') \vee (t_1 = v''' \vee t_4 = v'''')$ represented as a tree.

The construction proceeds inductively on the structure of a Boolean expression. Consider its tree representation (an example is shown in Figure 2): the *root* represents the entire expression, the *leaves* represent clauses of the form $t = v$ (note that "base" variables $x_{t,v}$ correspond to these clauses), and each *internal node* represents a disjunction, a conjunction, or a negation. To model these operations, we introduce new variables in the block Z and constrain them to match the Boolean value of the subexpression they represent. Constraining given Boolean variables with inequalities to ensure that they behave like Boolean operators can be done in a well-known manner [13].

For a *disjunction* of the form $\psi_{c_1 \vee c_2} = \psi_{c_1} \vee \psi_{c_2}$, and "raise" variables $r_1, r_2 \in \{0, 1\}$, such that

$$\psi_{c_1}(\boldsymbol{x}) = 1 \quad \Leftrightarrow \quad r_1 = 1$$
$$\psi_{c_2}(\boldsymbol{x}) = 1 \quad \Leftrightarrow \quad r_2 = 1,$$

we introduce a new variable $z \in \{0, 1\}$ and constrain it as follows:

$$\begin{aligned} r_1 \quad - z &\le 0 \\ r_2 - z &\le 0 \\ -r_1 - r_2 + z &\le 0 \end{aligned}$$

This ensures that $\psi_{c_1 \vee c_2}(\boldsymbol{x}) = 1$ if and only if $z = 1$.

The encoding of *conjunction* and *negation* can be done in similar manner [13]. Performing the described construction for every issue and raise condition allows us to fully incorporate the connections between the design choice and all raised issues.

Objective function. We can now define the coefficient vector $\boldsymbol{p} \in \mathbb{R}^n$ used in the objective function $\boldsymbol{p}^T\boldsymbol{x}$. We set the j-th element of $\boldsymbol{p}$ to the weight $w(c_i)$ of the issue c_i if the value of the j-th element of $\boldsymbol{x}$ is r_i (that is, if the issue c_i is eventually raised). In all other cases, we set the j-th element of $\boldsymbol{p}$ to 0. This gives us the following definition of

the objective function:

$$\boldsymbol{p}^T \boldsymbol{x} = \begin{pmatrix} \bar{\mathbf{0}} & w(c_1) & \cdots & w(c_{|C|}) & \bar{\mathbf{0}} \end{pmatrix} \begin{pmatrix} \bar{\mathbf{0}} \\ r_1 \\ \vdots \\ r_{|C|} \\ \bar{\mathbf{0}} \end{pmatrix}$$

The resulting sum exactly matches the additive inverse of the preferability of the design choice:

$$\boldsymbol{p}^T \boldsymbol{x} = \sum_{i=1}^{|C|} w(c_i) r_i = -\sigma_w(S_{\boldsymbol{x}}),$$

and, since the BLP problem is a minimization problem, an optimal solution for a BLP instance results in a design choice $S_{\boldsymbol{x}}$ of maximal preferability, as desired.

Polynomial reduction. We have thus presented the reduction of Spec-Dec-Opt to BLP. It remains to establish that this reduction can be done in polynomial time.

Indeed, for any given design template, the number of "base" (block `T1`) and "raise" (block `C1`) variables is bounded by $|T| \cdot |V| + |C|$. Constructing the blocks `T2L` and `T2R` requires adding two constraints per tweakable, and the constraints only employ "base" variables. Thus, the construction of the blocks `T2L`, `N1`, `N2`, `T1`, `C1`, and `T2R` can be done in polynomial time. The construction of the blocks `C2L`, `Z`, and `C2R` is highly dependent on the structure of raise conditions and individual issue activations. More precisely, the number of raised conditions and their complexity are bounded by the number of issues and complexity of issue activations, respectively. The number of logical operators per raise condition is bounded by the length of the condition expression itself, and stepping through them can be done with a depth-first approach. The amount of variables and constraints that need to be added per logical operator is constant. Constructing the coefficient vector $\boldsymbol{p}$ is bounded by the number of variables. This means that the entire process is a polynomial reduction.

We now showcase the described construction.

Example 3.5 (cf. Ex. 3.4 and Ex. 2.4)**.** For the two issues c_1 and c_2, we define:

$$\psi_{c_1} = (t_1 = v_{\text{false}})$$
$$\psi_{c_2} = (t_2 = v_{\text{false}} \land t_1 = v_{\text{true}})$$

Here we need to constrain the "base" variables to ensure that the "raise" variable r_1 matches $x_{t_1,\text{false}}$ and the "raise" variable r_2 matches the conjunction of $x_{t_2,\text{false}}$ and $x_{t_1,\text{true}}$. This gives us the following instance of the BLP problem:

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{t_1,v_{\text{true}}} \\ x_{t_1,v_{\text{false}}} \\ x_{t_2,v_{\text{true}}} \\ x_{t_2,v_{\text{false}}} \\ r_1 \\ r_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

with the vector $\boldsymbol{p}$ defined as $\boldsymbol{p}^T = (0, 0, 0, 0, w(c_1), w(c_2))$. □

3.3 Dealing with Uncertainty

In the Spec-Dec-Opt problem, one reasonable assumption is that a language designer cannot precisely specify the value of an issue severity, but is still able to specify a *range* for that value. One can then employ *robust optimization* techniques [14] that consider the most extreme parameter deviations, ensuring that the solution remains optimal (feasible) under all potential outcomes. One of such techniques is the *interval form of uncertainty* [15], where the worst-case scenario involves considering extreme values (i.e., the smallest lower bound value or the largest upper bound value) to test the robustness of the solution. Under yet another assumption—that a language designer is only able to specify a *set* of values for severity of an issue (rather than an exact value)—we can use *scenario-based* optimization techniques [14]. In such a setting, one considers multiple potential scenarios of uncertain parameters (in our case: severity values), optimizing the solution across these scenarios to minimize risk and ensure good performance on average or in the worst case.

4 Implementation

The DSL has been implemented using language workbench Langium [16]. In addition to the out-of-the-box functionality, the VSCode extension performs validation of DSL code, and reports corresponding errors and warnings (e.g., every tweakable must specify its default value, a condition must not reference itself, issues and conditions should not be unused, and so on).

Based on a DSL specification, the extension generates an interactive webpage that allows a language designer to experiment with the design space and observe the issues raised based on their design decisions. The user can ascribe weights to issues and automatically formulate an instance of BLP, which is solved by the SCIP solver [11] using the *PySCIPOpt* framework [12]. The solver uses a combination of different techniques, such as branch-and-cut, heuristics, Benders' decomposition [10, 11], to find optimal solutions to the *BLP* instance. These are then translated back to the design space and shown to the user.

The implementation is available at https://github.com/bldl/jspl.

5 Case Study

The design space of the ECMAScript *Records & Tuples* proposal [7] focuses on several aspects: zero values and their representations, equality testing (discussed in Section 2.2), type checking, object and collection behaviour, and wrapper operations. The individual design points (tweakables) and issues are summarized[16] in Tables 1 and 2.

Consider a visualization of the dependencies between tweakables in Figure 3. There are two branching points: tweakables t_{17} and t_{21} are both high out-degree nodes and represent critical—and potentially contentious—design points that significantly affect the overall design. Tweakables t_{14} and t_{16} are isolated from the rest of the design—as they are the only non-fixed tweakables dealing with NaN values in tuples.

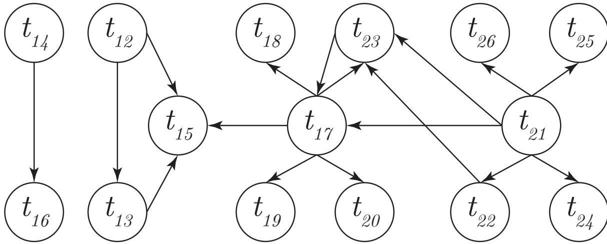

Figure 3. Dependencies between tweakables in Table 1.

Ascribing weight 1 to every issue, we have formulated an instance of the Spec-Dec-Opt problem. The corresponding BLP instance has 94 variables and 151 constraints, and any optimal solution has at least 3 issues: $\{c_1, c_7, c_{11}\}$, $\{c_1, c_7, c_{15}\}$, $\{c_1, c_7, c_{16}\}$, $\{c_1, c_8, c_{11}\}$, $\{c_1, c_8, c_{15}\}$, or $\{c_1, c_8, c_{16}\}$. It is interesting to observe that issue c_1 is raised if `Box` is not defined ("`undefined`")—this is decided in the contentious tweakable t_{21}, which affects several tweakables. Those dependent tweakables in their turn raise other issues if `Box` is defined ("`function`"). Moreover, defining `Box` raises issue c_{22}—which claims that the complexity of the language becomes increased—and the optimizer essentially "avoids" adding additional complexity to the language by preferring to keep `Box` undefined (and thus raising issue c_1).

Running the SCIP solver on this instance takes less than 20 ms on a Dell XPS 15 9570 machine (CPU: i5-8300H, RAM: 16 GB). We have experimentally formulated an instance of Spec-Dec-Opt with 1000 tweakables; the corresponding BLP instance was solved within 2.18 s by the SCIP solver.

[16]The complete specification of this proposal in our DSL is available at https://github.com/bldl/jspl/blob/main/examples/in/records_and_tuples.jspl.

6 Related Work

Using of mathematical optimization approaches, such as linear programming and constraint solving techniques, in *implementation* of programming languages and tooling for them, is far from novel. While it is impossible to give an exhaustive account for these use cases, we mention a few examples. Kästner and Langenbach [17] formulate instruction scheduling and register allocation as an integer linear program. Lozano et al. [18] discuss a similar setting in the context of LLVM [19]. Horton [20] considers the applications of (pseudo-)Boolean optimization for register allocation. Pandi et al. [21] study a certain kind of type inference as a joint optimization problem. Schenck et al. [22] use integer linear programming in the context of array programming language Futhark to find the minimum number of operations needed to transform an implicitly lifted expression into a well-typed program. Süslü and Csallner [23] employ Z3 solvers to perform symbolic execution during partial evaluation of JavaScript programs. Hague et al. [24] use Max-SAT solvers for minification of CSS stylesheets in a semantic-preserving way.

One can also speculatively imagine that many of the features in Integrated Development Environments (IDE), such as dependency resolution in plug-ins [25], can be presented as constraint satisfaction problems, and thus implemented as such. Dynamic layout adjustment in a projectional editor [26] (i.e., reflowing elements when new AST nodes are inserted) can be formulated in terms of a linear programming problem. Code refactoring can be considered from an optimization perspective: maximizing the refactoring benefit—be it reduced duplication or improved readability—for overlapping refactorings can be formulated as a Maximum Knapsack or a Maximum Independent Set problem.

While it is also possible to imagine the use of linear programming in the process of software language *design* and *engineering* (e.g., balancing the trade-off between expressiveness and simplificity, optimizing backward compatibility decisions by modelling the impact of breaking changes), the body of available literature seems very scarce. Gammaitoni et al. [27] use SAT solvers for verification of language specifications. We can tangentially mention Zhang et al. [28] who modeled Rust ownership, lifetimes, and mutability rules as a constraint-solving problem.

There is an extensive body of literature discussing software language design and evolution [29]. Classical works by Hoare [30], Brooks [31], McKeeman [32], and Wirth [33] set the foundations of language design as a discipline. Meyer [34] argues about the importance of aesthethics and elegance in language design *and* evolution. Zaytsev [35] proposes a language design toolkit to help software language designers make more informed and intentional design decisions. Coblenz et al. [36] advocate for an interdisciplinary

	tweakable name	*expression*	*possible values*
design space invariants (fixed tweakables)			
t_1	typeofArray	`typeof []`	`"object"`
t_2	typeofNan	`typeof NaN`	`"number"`
t_3	zeroTripleEqualsNegativZero	`+0 === -0`	`true`
t_4	zeroObjectIsNegativeZero	`Object.is(+0, -0)`	`false`
t_5	arrayWithNegativeZeroIncludesZero	`[-0].includes(+0)`	`true`
t_6	nanTripleEqualsNan	`NaN === NaN`	`true`
t_7	nanObjectIsNan	`Object.is(NaN, NaN)`	`true`
t_8	arrayWithNanIncludesNan	`[NaN].includes(NaN)`	`true`
t_9	arrayWithZeroTripleEqualsArrayWithZero	`[0] === [0]`	`false`
t_{10}	tupleWithZeroTripleEqualsTupleWithZero	`#[0] === #[0]`	`true`
t_{11}	objectIsFrozenTupleWithZero	`Object.isFrozen(#[0])`	`true`
individual design points (tweakables)			
t_{12}	storeNegativeZero	`Object.is(#[-0].at(0), -0)`	`true | false`
t_{13}	zerosAreTripleEqual	`#[+0] === #[-0]`	`true | false`
t_{14}	tupleNaNAreTripleEqual	`#[NaN] === #[NaN]`	`true | false`
t_{15}	tupleWithZeroObjectIsTupleWithNegativeZero	`Object.is(#[+0], #[-0])`	`true | false`
t_{16}	tupleWithNanObjectIsTupleWithNan	`Object.is(#[NaN], #[NaN])`	`true | false`
t_{17}	typeofTuple	`typeof #[]`	`"tuple" | "object"`
t_{18}	tupleWrappedInObjectTripleEqualsTuple	`Object(#[]) === #[]`	`true | false`
t_{19}	addingTupleToWeakSetThrows	`new WeakSet().add(#[])`	*ShouldThrow* or *ShouldSucceed*
t_{20}	tupleAsArgumentOfNewProxyThrows	`new Proxy(#[])`	*ShouldThrow* or *ShouldSucceed*
t_{21}	typeofBoxConstructor	`typeof Box`	`"undefined" | "function"`
t_{22}	typeofBoxInstance	`typeof Box()`	`"box" | "object"`
t_{23}	typeOfTupleWithBox	`typeof #[Box()]`	`"tuple" | "object"`
t_{24}	boxConstructorWithPrimitives	`Box(42)`	*ShouldThrow* or *ShouldSucceed*
t_{25}	addingTuplesWithBoxesToWeakSets	`new WeakSet().add(#[Box()])`	*ShouldThrow* or *ShouldSucceed*
t_{26}	tupleWithBoxAsArgumentForNewProxy	`new Proxy(#[Box()])`	*ShouldThrow* or *ShouldSucceed*

Table 1. Summary of the design space of the ECMAScript *Records & Tuples* proposal.

approach to programming language design. Curtis [37], Hanenberg [38], and Stefik et al. [39] discuss human factors in language design. Ousterhout [40] argues that "the decision-making process must be highly inclusive and it must allow consesus to emerge". Meyer [34] touches upon politics [41] of language evolution. Izquierdo and Cabot [1] give an overview of decision-making actors in the design and evolution of C++, Go, Java, Kotlin, PHP, Python, R, and Scala. Keertipati et al. [4] discuss in detail the decision-making process in Python and the divergence between the normative and the actual process. Sharma et al. [42] identify contentious topics in Python evolution. Kaschesky and Riedl [43] discuss the decision-making processes in Java evolution. Stroustrup talks about the decision-making process in C++ [44].

7 Conclusion

To the best of our knowledge, our approach is one of the first applications of linear programming methods to *fine-grained* design of *individual* language features of *widely adopted* programming languages.

We presented a formal definition of a design template and demonstrated its relevance and applicability by specifying the design space of a non-trivial feature in a widely adopted programming language. Again, to the best of our knowledge, this definition is novel, and the transformation described in Section 3 has not been previously explored in the literature.

The domain-specific language introduced in Section 2 captures language designer's intent when trying to specify decision points arising in feature design. Our implementation is an example of how tooling can support language design and evolution [27, 45], and it enables involving a wide audience of stakeholders in language evolution (cf. [36]).

One can also imagine that a specification in our DSL can be used to generate a "sandbox" IDE for the language with support of the proposal being defined. This tool, even if limited to that proposal, can be used to preview various design alternatives on existing codebases.

We note that a language design team can benefit from a precise textual specification of the design space by employing, for example, version control tools such as *diff* to track and compare language designs, collaborative editing tools to

	issue name	*short description*
c_1	`withoutBox`	shifting object storage complexity to external patterns and APIs
c_2	`typeofPowerfulObjectIsNotObject`	security risk from unexpected `typeof` behavior
c_3	`validWeakValue`	introduce inconsistency to `WeakSet` storage rules
c_4	`weakSetLeak`	potential for silent memory leaks
c_5	`slotSensitiveTypeof`	confusing `typeof` behavior based on contents
c_6	`confusingTypeof`	inconsistent `typeof` distinctions in tuples
c_7	`objectWrappers`	unexpected behavior when comparing tuples wrapped in objects
c_8	`objectWrapperInconsistency`	tuple object identity and cross-realm prototype linkage
c_9	`noBoxesInWeakSets`	performance limitations due to `WeakMap` restrictions
c_{10}	`unequalTupleNan`	checking for a value to not equal itself is used as check for NaN
c_{11}	`noNegativeZero`	loss of negative zero preservation
c_{12}	`impossibleEqualityOfZeros`	tuple equality inconsistency for zero values
c_{13}	`observableDifferentButIsEqual`	weakened guarantees of `Object.is` semantics
c_{14}	`nanNotIsNan`	inconsistencies in NaN equality semantics
c_{15}	`canNotAlwaysIntern`	interning limitations for performance optimization
c_{16}	`zerosNotTripleEqual`	unexpected behavior of `===`
c_{17}	`storingPrimitiveInBox`	complexity shift in `Box` usage
c_{18}	`noPrimitivesInBox`	ergonomic challenges in `Box` construction
c_{19}	`recordProxies`	limited utility of record proxies
c_{20}	`proxyThrowTypeofObject`	proxy constructor usability concerns
c_{21}	`differenceBetweenEqualityForTypeofObject`	discrepancy in equality for object-like values
c_{22}	`boxType`	increased language complexity with `Box` type
c_{23}	`objectsDontHaveWrappers`	redundant `ToObject` operation for objects
c_{24}	`tuplePrototypeEquality`	tuple prototype inconsistency across realms

Table 2. Summary of the issues in the design space of the ECMAScript *Records & Tuples* proposal.

evolve a specification, and various formal analyses to reason about a given design [27].

Solving the corresponding optimization problem allows the language design team to identify situations when no "perfect solution"—one with zero issues—exists. Using our approach, it is possible to identify the core parts of the design that have the most influence over the rest of the design—and this allows a language design team to focus on those. Indeed, if consensus could be made on one of those parts, it would help reduce the design space left to keep exploring.

Tweakables with only one possible value (cf. Table 1) can be considered as *language design invariants* (cf. [46]); they should be adhered to when new language features are designed. It is possible to imagine an extension of our DSL that supports importing specifications, thus reusing already defined fixed tweakables in multiple proposal specifications.

Impact

At its April 2025 plenary meeting[17], the TC39 committee decided to withdraw the *Records & Tuples* proposal. Our approach demonstrated that no solution could satisfy all design requirements, and, with no viable path to resolve these issues, withdrawal became the necessary outcome.

[17] https://github.com/tc39/notes/blob/main/meetings/2025-04/april-14.md

Acknowledgements

The authors express their gratitude to the anonymous reviewers for their valuable comments on the paper. The authors also thank Jaakko Järvi and Yan Passeniouk (University of Turku) for discussions about the paper, and Yulia Startsev (Mozilla) for facilitating the collaboration. M.B. acknowledges financial support from L. Meltzers Universitetsstiftelse.

References

[1] J. L. C. Izquierdo and J. Cabot, "Analysis and modeling of the governance in general programming languages," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019* (O. Nierstrasz, J. Gray, and B. C. d. S. Oliveira, eds.), pp. 179–183, ACM, 2019.

[2] E. International, "TC39 Process Document." https://tc39.es/process-document/, n.d. Accessed: 2025-04-25.

[3] ISOCPP, "The life of an ISO proposal: From "cool idea" to "international standard"." https://isocpp.org/std/the-life-of-an-iso-proposal, n.d. Accessed: 2025-04-25.

[4] S. Keertipati, S. A. Licorish, and B. T. R. Savarimuthu, "Exploring decision-making processes in Python," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, EASE 2016, Limerick, Ireland, June 01 - 03, 2016* (S. Beecham, B. A. Kitchenham, and S. G. MacDonell, eds.), pp. 43:1–43:10, ACM, 2016.

[5] W3C, "Process document." https://www.w3.org/policies/process/, n.d. Accessed: 2025-04-25.

[6] P. P. Repository, “Design by committee.” https://wiki.c2.com/?DesignByCommittee, 2011. Accessed: 2025-04-25.

[7] E. International, “TC39 proposal “Records & Tuples”.” https://github.com/tc39/proposal-record-tuple, n.d. Accessed: 2025-04-25.

[8] A. Claymore, “Record and Tuple Laboratory.” https://acutmore.github.io/record-tuple-laboratory/, n.d. Accessed: 2025-04-25.

[9] G. Nemhauser and L. Wolsey, *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988. Accessed: 2025-04-25.

[10] F. Clautiaux and I. Ljubić, “Last fifty years of integer linear programming: A focus on recent practical advances,” *European Journal of Operational Research*, vol. 324, no. 3, pp. 707–731, 2025.

[11] S. Bolusani *et al.*, “The SCIP Optimization Suite 9.0.” https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/, 2024. Accessed: 2025-04-25.

[12] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano, “PySCIPOpt: Mathematical programming in Python with the SCIP optimization suite,” in *Mathematical Software – ICMS 2016* (G.-M. Greuel, T. Koch, P. Paule, and A. Sommese, eds.), (Cham), pp. 301–307, Springer International Publishing, 2016.

[13] “MIP formulations and linearizations: Quick reference.” , 2017. Accessed: 2025-04-25.

[14] P. Kouvelis and G. Yu, *Robust Discrete Optimization and Its Applications*, vol. 14 of *Nonconvex Optimization and Its Applications*. New York, NY: Springer, 1997.

[15] A. Kasperski, *Discrete Optimization with Interval Data: Minmax Regret*, vol. 2686 of *Lecture Notes in Computer Science*. Springer, 2003. Accessed: 2025-04-25.

[16] T. GmbH, “Langium.” https://langium.org, n.d. Accessed: 2025-04-25.

[17] D. Kästner and M. Langenbach, “Code optimization by integer linear programming,” in *Compiler Construction, 8th International Conference, CC'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings* (S. Jähnichen, ed.), vol. 1575 of *Lecture Notes in Computer Science*, pp. 122–136, Springer, 1999.

[18] R. C. Lozano, M. Carlsson, G. H. Blindell, and C. Schulte, “Combinatorial register allocation and instruction scheduling,” *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 3, pp. 17:1–17:53, 2019.

[19] C. Lattner and V. S. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pp. 75–88, IEEE Computer Society, 2004.

[20] M. R. Horton, “Register allocation by pseudo-boolean optimization (thesis).” https://dash.harvard.edu/server/api/core/bitstreams/9e2695c8-954a-486b-b529-93ff1f6c9f84/content, 2020. Accessed: 2025-04-25.

[21] E. V. Pandi, E. T. Barr, A. D. Gordon, and C. Sutton, “Type inference as optimization,” in *Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS)*, 2021. Accessed: 2025-04-25.

[22] R. Schenck, N. H. Hinnerskov, T. Henriksen, M. Madsen, and M. Elsman, “AUTOMAP: inferring rank-polymorphic function applications with integer linear programming,” *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, pp. 1787–1813, 2024.

[23] S. Süslü and C. Csallner, “SPEjs: a symbolic partial evaluator for JavaScript,” in *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, A-Mobile 2018, (New York, NY, USA), p. 7–12, Association for Computing Machinery, 2018.

[24] M. Hague, A. W. Lin, and C.-D. Hong, “CSS minification via constraint solving,” *ACM Trans. Program. Lang. Syst.*, vol. 41, June 2019.

[25] A. Kornstädt and E. Reiswich, “Composing systems with Eclipse Rich Client Platform plug-ins,” *IEEE Softw.*, vol. 27, no. 6, pp. 78–81, 2010.

[26] M. Völter, J. Siegmund, T. Berger, and B. Kolb, “Towards user-friendly projectional editors,” in *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings* (B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, eds.), vol. 8706 of *Lecture Notes in Computer Science*, pp. 41–61, Springer, 2014.

[27] L. Gammaitoni, P. Kelsen, and C. Glodt, “Designing languages using Lightning,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015* (R. F. Paige, D. D. Ruscio, and M. Völter, eds.), pp. 77–82, ACM, 2015.

[28] H. Zhang, C. David, Y. Yu, and M. Wang, “Ownership guided C to Rust translation,” in *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III* (C. Enea and A. Lal, eds.), vol. 13966 of *Lecture Notes in Computer Science*, pp. 459–482, Springer, 2023.

[29] J. Favre, “Languages evolve too! Changing the software time scale,” in *8th International Workshop on Principles of Software Evolution (IWPSE 2005), 5-7 September 2005, Lisbon, Portugal*, pp. 33–44, IEEE Computer Society, 2005.

[30] C. Hoare, “Hints on programming language design,” in *State of the Art Report 20: Computer Systems Reliability* (C. Bunyan, ed.), pp. 505–534, Pergamon/Infotech, 1974.

[31] F. P. Brooks, *Keynote address: language design as design*, p. 4–16. New York, NY, USA: Association for Computing Machinery, 1996.

[32] W. M. McKeeman, *Programming Language Design*, pp. 514–524. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974.

[33] N. Wirth, “On the design of programming languages,” in *Proceedings of the IFIP Congress*, pp. 386–393, 1974. Accessed: 2025-04-25.

[34] B. Meyer, “Principles of language design and evolution,” in *Millennial Perspectives in Computer Science*, pp. 229–246, Palgrave, 2000. Accessed: 2025-04-25.

[35] V. Zaytsev, “Language design with intent,” in *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*, pp. 45–52, IEEE Computer Society, 2017.

[36] M. J. Coblenz, J. Aldrich, B. A. Myers, and J. Sunshine, “Interdisciplinary programming language design,” in *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2018, Boston, MA, USA, November 7-8, 2018* (E. G. Boix and R. P. Gabriel, eds.), pp. 133–146, ACM, 2018.

[37] B. Curtis, “A review of human factors research on programming languages and specifications,” in *Proceedings of the 1982 Conference on Human Factors in Computing Systems, CHI 1982, Gaithersburg, Maryland, USA, March 15-17, 1982* (J. A. Nichols and M. L. Schneider, eds.), pp. 212–218, ACM, 1982.

[38] S. Hanenberg, “Faith, hope, and love: an essay on software science's neglect of human factors,” in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA* (W. R. Cook, S. Clarke, and M. C. Rinard, eds.), pp. 933–946, ACM, 2010.

[39] A. Stefik, S. Hanenberg, M. McKenney, A. A. Andrews, S. K. Yellanki, and S. Siebert, “What is the foundation of evidence of human factors decisions in language design? An empirical study on programming language workshops,” in *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014* (C. K. Roy, A. Begel, and L. Moonen, eds.), pp. 223–231, ACM, 2014.

[40] J. Ousterhout, “Open decision-making.” https://web.stanford.edu/~ouster/cgi-bin/decisions.php, 2021. Accessed: 2025-04-25.

[41] M. B. H. Weiss, “The standards development process: a view from political theory,” *ACM Stand.*, vol. 1, no. 2, pp. 35–41, 1993.

[42] P. Sharma, B. T. R. Savarimuthu, N. Stanger, S. A. Licorish, and A. Rainer, "Investigating developers' email discussions during decision-making in Python language evolution," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, EASE 2017, Karlskrona, Sweden, June 15-16, 2017* (E. Mendes, S. Counsell, and K. Petersen, eds.), pp. 286–291, ACM, 2017.

[43] M. Kaschesky and R. Riedl, "Top-level decisions through public deliberation on the internet: evidence from the evolution of Java governance," in *Proceedings of the 10th Annual International Conference on Digital Government Research, Partnerships for Public Innovation, DG.O 2009, Puebla, Mexico, May 17-20, 2009* (S. A. Chun, R. Sandoval, and P. M. Regan, eds.), vol. 390 of *ACM International Conference Proceeding Series*, pp. 42–55, Digital Government Research Center, 2009.

[44] B. Stroustrup, "Evolving a language in and for the real world: C++ 1991-2006," in *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007* (B. G. Ryder and B. Hailpern, eds.), pp. 1–59, ACM, 2007.

[45] E. Visser, G. Wachsmuth, A. P. Tolmach, P. Neron, V. A. Vergu, A. Passalaqua, and G. Konat, "A language designer's workbench: A one-stop-shop for implementation and verification of language designs," in *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014* (A. P. Black, S. Krishnamurthi, B. Bruegge, and J. N. Ruskiewicz, eds.), pp. 95–111, ACM, 2014.

[46] Y. Startsev, "Documenting invariants and design principles." https://github.com/codehag/documenting-invariants/blob/main/known_invariants.md, n.d. Accessed: 2025-04-25.

[47] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, 2013.

A Appendix: Proof of Theorem 3.3

The proof is by reduction of SAT to Spec-Dec-Opt.

> **Boolean Satisfiability Problem (SAT)** [47]: Does a given Boolean formula φ over atomic variables have a satisfying variable assignment?

We demonstrate how an arbitrary Boolean formula can be translated to an instance of Spec-Dec-Opt, and show that an optimal design choice for this instance can be used to determine whether or not the original formula is satisfiable.

Consider a Boolean formula φ over the variables $\{x_1, \dots, x_\ell\}$.

For each variable x_i, construct a tweakable t_i with two possible values true and false and expression $\square$ (which is irrelevant for the construction):

$$\begin{aligned} t_i ::= \ & \underline{\texttt{select}} \ \square \ \underline{\texttt{of}} \\ & \underline{\texttt{case}} \ \text{true} : \varnothing \\ & \underline{\texttt{case}} \ \text{false} : \varnothing \end{aligned}$$

We define an issue c and a tweakable t_{SAT} as follows:

$$\begin{aligned} t_{SAT} ::= \ & \underline{\texttt{select}} \ \square \ \underline{\texttt{of}} \\ & \underline{\texttt{case}} \ \text{true} : \{\neg b_\varphi \Rightarrow c\} \end{aligned}$$

Here Boolean expression b_φ is constructed by replacing each variable x_i in the original formula φ with a term $(t_i = \text{true})$.

Now an instance of Spec-Dec-Opt can be defined by a tuple $\langle C, A, T \rangle$, where the set of issues is $C = \{c\}$, the set of issue activations is $A = \{\neg b_\varphi \Rightarrow c\}$, and the set of tweakables is $T = \{t_{SAT}, t_1, \dots, t_\ell\}$. We set the weight of the only issue c to be 1, that is, the severity ascription is $w(c) = 1$.

Assume we have a solution to this instance of Spec-Dec-Opt and S is an optimal design choice. We can define an interpretation I_S such that $I_S(x_i) = 1$ if and only if $S(t_i) =$ true. We now need to establish that

$$\varphi \text{ is satisfiable} \quad \Leftrightarrow \quad I_S(\varphi) = 1 \quad \Leftrightarrow \quad \sigma_w(S) = 0$$

Assume φ is satisfiable. By construction, there exists a design choice S' for which b_φ is evaluated to true. For S', the issue activation $(\neg b_\varphi \Rightarrow c)$ is not activated and the issue c is not raised. Hence, the optimal preferability is $\sigma_w(S') = 0$. Since S is also an optimal solution, this means that $\sigma_w(S) = \sigma_w(S') = 0$. We can also establish that $I_S(\varphi) = 1$.

If φ is unsatisfiable, then $I_S(\varphi) = 0$ and the value of b_φ is false. Hence, the issue activation $(\neg b_\varphi \Rightarrow c)$ is activated and the issue c is raised in the design choice S. Thus, $\sigma_w(S) = -w(c) = -1$.

Thus, determining whether φ is satisfiable can be done by constructing an interpretation I_S and checking whether or not it satisfies φ.

It remains now to establish that the reduction takes polynomial time. Indeed, constructing tweakables $t_1, \dots, t_\ell$ takes a constant amount of operations. Constructing the expression b_φ is done by stepping through φ in polynomial time.

Thus, SAT $\leq_p$ Spec-Dec-Opt.

Note that SAT is a *decision* problem, whereas Spec-Dec-Opt is an *optimization* problem. In the reduction, we demonstrated that the corresponding decision version of Spec-Dec-Opt—namely, *finding a design choice that raises no issues*—can solve SAT. Because SAT is **NP**-hard [47], and since optimization problems are always at least as hard as their corresponding decision problems, Spec-Dec-Opt is **NP**-hard. $\square$