**DOI: 10.15514/ISPRAS-2025-37(6)-56**

# On a Relational Code Model Description

[1] *D.E. Duzhinskii, ORCID: 0009-0009-0164-7547 <dduzhinsky@ya.ru>*
[2] *D.Yu. Boulytchev, ORCID: 0000-0001-8363-7143 <dboulytchev@math.spbu.ru>*
[1] *LLC "Yandex.Technologies",*
*16, Lev Tolstoy st., Moscow, 119021, Russia.*
[2] *Saint-Petersburg State University,*
*7/9, Universitetskaya emb., Saint-Petersburg, 199034, Russia.*

**Abstract.** We present an approach for systematically developing code models for programming languages based on the idea of relational specification DSL. The DSL allows to describe the entities of programming language, relations between these entities, and form queries for these relations. Besides simple information retrieval the query language can be used to specify derived relations which makes it possible to saturate an initial code model with usecase-specific intermediate representations defined in a declarative form. The DSL is strongly typed which guarantees the preservation of code model invariants. We introduce the language, its dynamic and static semantics, describe prototype implementation and evaluate certain scenarios of its utilization.

**Keywords:** code model; code analysis; relational programming.

**For citation:** Duzhinskii D.E., Boulytchev D.Yu. On a Relational Code Model. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 4, 2025, pp. 159-174. DOI: 10.15514/ISPRAS-2025-37(6)-56.

# Об описании реляционной модели кода

[1] *Д.Е. Дужинский, ORCID: 0009-0009-0164-7547 <dduzhinsky@ya.ru>*
[2] *Д.Ю. Булычев, ORCID: 0000-0001-8363-7143 <dboulytchev@math.spbu.ru>*

[1] *ООО "Яндекс.Технологии",*
*Россия, 119021, г. Москва, ул. Льва Толстого, д. 16.*

[2] *Санкт-Петербургский Государственный Университет,*
*Россия, 199034, г. Санкт-Петербург, Университетская наб., д. 7/9.*

**Аннотация.** Мы описываем подход к систематической разработке моделей кода языков программирования. Наш подход основан на идее использования реляционного предметно-ориентированного языка. Этот язык позволяет описывать сущности языков программирования, отношения между этими сущностями, а также запросы к этим отношениям. Помимо собственно извлечения информации язык запросов позволяет в декларативной форме задавать производные отношения, что открывает возможность насыщать исходную модель кода представлениями, специфическими для конкретных сценариев использования. Предлагаемый нами язык строго-типизирован, что предоставляет гарантии сохранения инвариантов модели кода. Мы описываем сам язык, его статическую и динамическую семантики, прототипную реализацию, а также представляем результаты апробации для некоторых сценариев его использования.

**Ключевые слова:** модель кода; анализ кода; реляционное программирование.

**Для цитирования:** Дужинский Д.Е., Булычев Д.Ю. О реляционной модели кода. Труды ИСП РАН, том 37, вып. 6, часть 4, 2025 г., стр. 159–174 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)-56.

## *1. Introduction*

No production development process nowadays can be managed without an extensive use of software development automation tools which assist developers in various ways by providing navigation through the code base, performing refactorings and other program transformations, support testing, continuous integration, collaborative development, etc. All these tools require a certain intermediate program representation, – *code model* – to be built and maintained in sync with the source code. However, different tools require different aspects of the code base to be reflected by a code model. Thus, as a rule, multiple intermediate representations are used to fit the needs of the variety of tools, and implementing the conversion between them manually is tedious and error-prone.

In this report we present a certain approach for describing code models. We introduce a DSL which specifies code models in terms of entities (notions of the language being described) and relations between them. This approach allows us to form queries to retrieve the information about programs; moreover, we introduce a mechanism to describe derived relations based on predefined ones. Thus, we make it possible to declaratively describe various program representations and conversions between them. Our DSL is typed which guarantees the preservation of certain important invariants which otherwise should have to be enforced in an *ad-hoc* manner.

The paper is organized as follows. In Section 2 we give an observable example of a code model described with the aid of our language. In Section 3 we formally describe relational expression language, its syntax, static and dynamic semantics, and formulate the soundness property. In Section 4 we give a short description of a prototype implementation and its evaluation for the problem of declarative control flow graph construction from syntactic structure of a program. Section 5 briefly observes related work, and Section 6 concludes.

Our work is still on early stage, but we consider the preliminary results inspiring.

## *2. Code Model Description Example*

In this section we gradually introduce the notions of our code model description DSL and present a few cases of its utilization. We do not use any concrete programming language to describe the model

for (it would be too verbose) but rather use various features from well-known languages to make the example small and observable.

As we already mentioned we consider code model as a collection of entities and relations between them. There is, however, an important observation we need to discuss first: we have to put a clear distinction between code model as a *meta*-description and *concrete* code model of a program as an instance of this meta-description. In the former sense a code model describes entities and relations, which serve as *types* of entities and relations for code models in the latter sense. Since in our DSL all entities and relations are, in turn, typed, unrestricted usage of notion "type" would be misleading since it may not be clear from the context which "type" we mean: the type of entity/relation in the sense of our DSL or entity/relation *itself* as a "type" of elements of a concrete code model for a given program. Thus, we will use the term "type" exclusively in the sense of our DSL, and will refer to elements of a concrete code model as "instances". Thus, entities/relations have instances (for which they serve as "types"), and are, in turn, typed (these types serve as types of types of instances).

So, we start from a simple entity specification:

```
entity Named {name: String}
```

Here an entity `Named` is introduced; each instance of entity `Named` holds an attribute "`name`" of type `String`. We assume a conventional set of primitive types available to use as types of entity's attributes. Similarly,

```
entity Visible {visibility: Public | Private | Protected}
```

introduces another entity, each instance of which holds one attribute of specified enum type with a specified name. Given these two definitions we can define "derived" entities

```
entity Class  : Named;
entity Method : Named, Visible;
```

which designate abstractions for classes and methods in some object-oriented language. Entities "`Class`" and "`Method`" are sub-entities of "`Named`" and "`Named`" and "`Visible`" respectively. Thus, the set of entity types is equipped with a subtyping relation. In this concrete example the usage of subtyping might look vacuous, but later we will present a more justified case. We now can define a relation "method"

```
method: Class, Method*;
```

which relates each class with a (possibly empty) set of its methods. A relation definition consists of a name for this relation (here "`method`") and specification of its *endpoints*. An endpoint is an entity name equipped with an optional *arity modifier* which can be used to specify conventional properties of relation such as one-to-one, one-to-many etc. In this particular case the relation is one-to-many since no specifier is used for its "left" set (hence "one" is assumed) and "*" is used for its "right" set (which means "zero-or-more"; the others are "?" ("zero-or-one") and "+" ("one-or-more")). The relation "`method`" is *atomic* since we do not specify its definition, only its type. This means that the concrete elements of this relation in any instance of the model have to be provided in an *ad-hoc* manner by some external tool; the model itself does not prescribe any way to calculate this relation. Thus, the creation of an instance of the model consists of creating a set of entity "Class" instances, entity "Method" instances, and putting them (somehow) in a one-to-many correspondence via relation "method". In reality, our tool generates all needed APIs for the language of model implementation which allow to perform all these steps. In order to make it possible to present a more interesting example we need a little more entities and relations in our model:

```
entity Stmt;
entity SeqStmt        : Stmt;
entity MethodCallStmt : Stmt;

head: SeqStmt, Stmt;
tail: SeqStmt, Stmt;
```

```
method: MethodCallStmt, Method;
body  : Method, Stmt;
```

Here we defined a new entity "`Stmt`" which serves as an abstraction for statements in some language, and two its sub-entities, – "`SeqStmt`" and "`MethodCallStmt`" – which can be seen as abstractions for sequential composition and method calls, respectively. We also added a few atomic relations: "`head`", which relates each sequential statement to its first component, "`tail`", which does the same for the second, "`method`", which relates a method call statement to the callee, and "`body`", which relates a method to its body. Note, in order to build an instance of relation "`method`" one needs to resolve the method's name (conventionally used in a call statement) into the definition of corresponding method. This may or may not be desirable in a concrete case; we use it just to make it clear that our approach does not necessarily operate of a purely syntactic level of programs' representation.

Given these definitions we can operate on instances of this model; each instance would probably contain some classes, some methods of these classes, and bodies of these methods. Note, the typed nature of our DSL guarantees the preservation of certain invariants: for example, it is guaranteed that any method has (exactly one) body, and that sequential statement is composed of exactly two components. We also can make an observation that the instances of this model can contain some *other* statements besides sequential composition and method calls since there can be some instances of entity "`Stmt`" which are not instances of entities "`SeqStmt`" and "`MethodCallStmt`"; thus, we can control the "granularity" of models by specifying only those part of the language of interest this model deals with.

Given an instance of this model we can, of course, retrieve its content by queries like in ordinary databases. What is more interesting we can define *derived* relations using atomic ones as starting point:

```
methodCalls: MethodCallStmt, Method  = method;
methodCalls: SeqStmt,        Method+ = (head + tail) . methodCalls;
```

Here, for the first time, we defined some relations with a non-empty right-hand side. These two lines actually by case-analysis define a single relation "`methodCalls`" with the type

```
Stmt, Method*
```

This requires some explanations. First, since in our model there is a unique common supertype for entities "`MethodCallStmt`" and "`SeqStmt`", namely, "`Stmt`", the actual derived type for the "left" set of "`methodCalls`" is "`Stmt`". Then, the first equation tells us that if a statement, for which we are looking for a callee, is actually a method call statement, then the callee is the callee of this method call. Obviously, when we equate relations we equate their types as well, thus the type for this branch of "`methodCalls`" definition should be equal the type of "`method`", which is "`MethodCallStmt, Method`".

The second equation is more interesting. This time we deal with sequential statement. In order to find all callees we take the head and the tail of this statement (which gives us *two* other statements), and then find the set of callees in each of them using the same "`methodCalls`" relation. Since there are two statements for which we are looking up for callees, there can be more than one callee found, hence we type this case of relation "`methodCalls`" definition as

```
Stmt, Method+
```

Finally, as there can be other statements besides method calls and sequential composition for which our definition would not deliver any results, we have to generalize the type once more, thus obtaining

```
Stmt, Method*
```

as it was promised. Note, the definition of "`methodCalls`" is recursive; we could alternatively define it as

```
methodCalls: Stmt, Method* = fix [r → method + (head + tail) . r];
```

where "**`fix`**" designates the least fixed-point operator for relations.

Now, with "`methodCalls`" defined, we can define the following derived relation

```
calls: Method, Method* = body . methodCalls;
```

which relates each method to all methods it calls; on other word, the call graph.

It worth mentioning that since we deal with relations we can do queries in both "directions"; thus, the inverse relation "`calls`" (in a concrete syntax "`calls'`") relates each method with all its call sites.

We summarize the contents of this section as follows:

- the DSL we introduce allows to describe a typed relational model of a code being manipulated;
- an instance of a model has to be built by external tools (parsers, compilers, etc.) using a model-specific API we generate from the model description;
- an instance of a model can be queried for contents like a database;
- besides atomic relations which are provided by external tools the DSL allows for describing derived relations which can be used in various scenarios to build relations not initially provided by external tools; these derived relations are typed as well and can be recursive.

## 3. *Code Model Specification DSL*

In this section we provide a formal description for the subset of code model specification DSL. As we've seen in the previous section our DSL describes entities and relations, and allows to specify derived relations in terms of other relations (atomic or derived) using a set of relational operators and a fixpoint operator. The descriptions of entities and atomic relations hardly require a formal treatment. We only assume that the set of entities is equipped with a sub-entity relation which is required to be a partial order. On the other hand, the static and dynamic semantics of the relational expression language requires formal specification and justification of soundness, which we provide in this section.

The syntax of relational expression language is shown in the Fig. 1. The primitive form of an expression is atomic relation, which is just a name. More complex expressions can be built from simpler ones using a few relational binary or unary operators or a fixed point operator. In the actual implementation we additionally use a few constant relations (in particular, reflexive and empty ones), but here we omit them for simplicity since their semantics is trivial. We describe the semantics of these operators in the following section.

$$
\begin{array}{rcll}
\mathcal{R} & = & r & \text{— atomic relation} \\
 & & \mathcal{R}^{-1} & \text{— inversion} \\
 & & \mathcal{R} \circ \mathcal{R} & \text{— composition} \\
 & & \mathcal{R} \cup \mathcal{R} & \text{— union} \\
 & & \mathcal{R} \mid \mathcal{R} & \text{— choice} \\
 & & \mathbf{fix}\ [r \mapsto \mathcal{R}] & \text{— fixed point}
\end{array}
$$

*Fig. 1. The syntax of relation expressions.*

### 3.1. *Evaluation Semantics*

Now we give an evaluation semantics for the relational expressions. First, we introduce *evaluation environments*, Δ:

$$
\begin{array}{rcll}
\Delta & = & \varnothing & \text{— empty environment} \\
 & & (r,\ S \subseteq X \times Y)\Delta & \text{— } X, Y \subseteq \Omega
\end{array}
$$

We fix some finite universal set Ω and consider all relations to be binary relations over Ω. The finality requirement follows from a trivial observation that all instances of all code models in practice consist of a finite number of entities' instances.

An evaluation environment maps atomic relation names to some binary relations. The semantics of $[\![E]\!]\Delta$ relational expression *E* is a relation defined as follows (Fig. 2).

$$
\begin{array}{rcl}
[\![r]\!]((r,\, S)\Delta) & = & S \\
[\![R^{-1}]\!]\Delta & = & \{(y,\, x) \mid (x,\, y) \in [\![R]\!]\Delta\} \\
[\![R \circ Q]\!]\Delta & = & \{(x,\, z) \mid (x,\, y) \in [\![R]\!]\Delta \wedge (y,\, z) \in [\![Q]\!]\Delta\} \\
[\![R \cup Q]\!]\Delta & = & [\![R]\!]\Delta \cup [\![Q]\!]\Delta \\
[\![R \mid Q]\!]\Delta & = & \{(x,\, y) \mid (x,\, y) \in [\![R]\!]\Delta \vee (\forall z\,.\,(x,\, z) \notin [\![R]\!]\Delta \wedge (x,\, y) \in [\![Q]\!]\Delta)\} \\
[\![\mathbf{fix}\ [r \mapsto R]]\!]\Delta & = & \bigcup_{n=0}^{\infty} \mathcal{R}_n, \text{ where } \mathcal{R}_0 = [\![R^n]\!]((r,\, \varnothing)\Delta),\ \mathcal{R}_{n+1} = [\![R^n]\!]((r,\, \mathcal{R}_n)\Delta)
\end{array}
$$

*Fig. 2. The semantics of relation expressions.*

We comment on this definition. For an atomic relation *r* the result of its evaluation is exactly the relation this atomic relation is associated with in corresponding evaluation environment; if there is no association the result of evaluation is undefined. The inversion operator ($^{-1}$) first evaluates its argument and then (if the result is defined) performs an inversion. Product/join operator ($\circ$) evaluates both its arguments and then builds their product. Left-biased choice operator builds a set of pairs such that a pair *(x, y)* belongs to this set iff either *(x, y)* belongs to its left operand or there is no *z* that *(x, z)* belongs to the left operand, but *(x, y)* belongs to the right one. Finally, the fixpoint operator is evaluated as a (potentially infinite) union of accumulating iterations of corresponding function, starting from an empty relation.

The only thing which requires justification is the semantics of the fixpoint operator. However, it is obvious that the set of all relations over Ω forms a complete lattice w.r.t. conventional set-theoretic union and intersection, and that all operators we use are monotonic in all their arguments. Since all relations are finite, the semantics of operator `fix` delivers the semantics of the least fixed point, which always exists.

## 3.2. *Static Semantics*

In the previous section we defined an evaluation semantics of relational operators; however the evaluation rules did not put any constraints on the relations being operated and, thus, no properties of the results (for example, non-emptiness) could be assessed. In this section we introduce the notion of type for a relation, provide typing rules for relational expressions, and prove the soundness property.

First, we fix a set of *entities E* equipped with a binary relation, which we require to be a partial order. Then, we introduce a notion of *arity A={?, +, *, 1}*. We call and *endpoint* a pair $\varepsilon^{\alpha}$, where ε is an entity, α is an arity. We extend the subtyping relation for endpoints as well as follows:

$$
\frac{\epsilon_1 \sqsubseteq \epsilon_2}{\epsilon_1^1 \sqsubseteq \epsilon_2^?} \qquad \frac{\epsilon_1 \sqsubseteq \epsilon_2}{\epsilon_1^+ \sqsubseteq \epsilon_2^*} \qquad \frac{\epsilon_1 \sqsubseteq \epsilon_2}{\epsilon_1^a \sqsubseteq \epsilon_2^a}
$$

It can be proven by case analysis that this extension is correct, i.e., that the extended relation is reflexive, antisymmetric, and transitive.

Finally, we define a type of relation to be a pair of endpoints and extend the subtyping relation for endpoints to subtyping relation for relation types component-wise:

$$
\frac{\epsilon_1^{\alpha_1} \sqsubseteq \epsilon_2^{\alpha_2},\ \epsilon_3^{\alpha_3} \sqsubseteq \epsilon_4^{\alpha_4}}{\epsilon_1^{\alpha_1}\epsilon_3^{\alpha_3} \sqsubseteq \epsilon_2^{\alpha_2}\epsilon_4^{\alpha_4}}
$$

Obviously, this extension preserves the properties of partial order.

In order to specify typing rules, we need to track down how arities of endpoints transform when a certain operation is performed over relations. For this we introduce three additional operators for arities (Fig. 3): sum ("$\oplus$"), product (" ") and choice $\otimes$ ("|").

| $\oplus$ | 1 | ? | + | * |
|---|---|---|---|---|
| 1 | + | + | + | + |
| ? | + | * | + | * |
| + | + | + | + | + |
| * | + | * | + | * |

| $\otimes$ | 1 | ? | + | * |
|---|---|---|---|---|
| 1 | 1 | ? | + | * |
| ? | ? | ? | * | * |
| + | + | * | + | * |
| * | * | * | * | * |

| \| | 1 | ? | + | * |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| ? | 1 | * | + | * |
| + | + | + | + | + |
| * | + | * | + | * |

*Fig. 3. Operations on arities.*

Now we are ready to give the typing rules. First, we define *typing environment* Γ:

$$\begin{array}{rcll} \Gamma & = & \varnothing & \text{— empty environment} \\ & & (r,\, t)\Gamma & \text{— } r \text{ — atomic relation, } t \text{ —type} \end{array}$$

Typing environments map atomic relations to their types; we also need a notion of a domain of a typing environment *dom* (Γ) which is just a set of bound atomic relation names in Γ:

$$\begin{array}{rcl} dom\,(\varnothing) & = & \varnothing \\ dom\,((r,\, t)\Gamma) & = & \{r\} \cup dom\,(\Gamma) \end{array}$$

The typing rules themselves are shown in the Fig. 4.

$$(r,\, t)\Gamma \vdash r : t \qquad \text{[T-Atom]}$$

$$\frac{(r,\, t)\Gamma \vdash \mathcal{R} : t_1, t_1 \sqsubseteq t_2}{(r,\, t)\Gamma \vdash \mathcal{R} : t_2} \qquad \text{[T-Subsumption]}$$

$$\frac{\Gamma \vdash \mathcal{R} : e_1 e_2}{\Gamma \vdash \mathcal{R}^{-1} : e_2 e_1} \qquad \text{[T-Inv]}$$

$$\frac{\Gamma \vdash \mathcal{R}_1 : \epsilon_1^{a_1}\epsilon_2^{a_2},\ \mathcal{R}_2 : \epsilon_2^{\hat{a_2}}\epsilon_3^{a_3}}{\Gamma \vdash \mathcal{R}_1 \circ \mathcal{R}_2 : \epsilon_1^{a_1}\epsilon_3^{a_2 \otimes \hat{a_2} \oplus a_3}} \qquad \text{[T-Comp]}$$

$$\frac{\Gamma \vdash \mathcal{R}_1 : \epsilon_1^{a_1}\epsilon_2^{a_2},\ \mathcal{R}_2 : \epsilon_1^{\hat{a_1}}\epsilon_2^{\hat{a_2}}}{\Gamma \vdash \mathcal{R}_1 \cup \mathcal{R}_2 : \epsilon_1^{a_1 \oplus \hat{a_1}}\epsilon_2^{a_2 \oplus \hat{a_2}}} \qquad \text{[T-Union]}$$

$$\frac{\Gamma \vdash \mathcal{R}_1 : \epsilon_1^{a_1}\epsilon_2^{a_2},\ \mathcal{R}_2 : \epsilon_1^{\hat{a_1}}\epsilon_2^{\hat{a_2}}}{\Gamma \vdash \mathcal{R}_1 \mid \mathcal{R}_2 : \epsilon_1^{a_1 | \hat{a_1}}\epsilon_2^{a_2 | \hat{a_2}}} \qquad \text{[T-Choice]}$$

$$\frac{(r,\, t)\Gamma \vdash R : t,\ r \notin dom\,(\Gamma)}{\Gamma \vdash \mathbf{fix}\ [r \mapsto R] : t} \qquad \text{[T-Fix]}$$

$$\frac{\Gamma \vdash \mathcal{R} : t_1,\ t_2 \sqsubseteq t_1}{\Gamma \vdash (\mathcal{R} : t_2) : t_2} \qquad \text{[T-Downcast]}$$

*Fig. 4. Typing rules.*

The first rule just assigns a type to an atomic relation according to the typing environment. The next one is a conventional typing rule for type systems with subtyping [1], which tells us that every object can be implicitly "casted" to its supertype. The next four rules, [T-Inv], [T-Comp], [T-Union], and [T-Choice] accurately propagate types according to corresponding relational operation. Rule [T-Fix] handles fixed point operator case; note the requirement for *r* to not to be bound in the environment. This requirement can always be satisfied by renaming the variables over which the fixpoint is taken.

Finally, the last rule, [T-Downcast], deals with the case when we have an explicit type specification for a relation. Note, we do not have dynamic semantics for this construct yet. We will address this aspect in the following section.

### 3.3. *Soundness*

The soundness property informally tells us that if a relational expression is typed in some typing environment, then, being evaluated for atomic relations obeying the constraints this environment puts of these relations, it would deliver a result which obeys the constraints prescribed by the type of the relation. In order to establish the soundness, we need to relate types and relations; we also need to enrich the evaluation semantics with the rule for type ascription. We did not do this yet since it relies on a correspondence between types and relations which we are going to build right now.

Let $\Omega$ be a finite set. We first consider a *valuation* of entities to be a function $\psi$: $E \to 2^{\Omega}$ such that $\varphi(\varepsilon_1) \subseteq \varphi(\varepsilon_2)$ whenever $\varepsilon_1 \sqsubseteq \varepsilon_2$ for all entities $\varepsilon_1, \varepsilon_2$. Thus, we interpret entities by subsets of $\Omega$ and sub-entity relation as a subset relation.

Given an arity $\alpha$ we interpret it as a predicate on sets $\hat{\alpha}$ as follows:

$$\begin{array}{rcl} \hat{1}\ (X) & = & \exists x \in X \wedge (\forall y \in X \,.\, y = x) \\ \hat{?}\ (X) & = & \forall x, y \in X \,.\, x = y \\ \hat{+}\ (X) & = & \exists x, x \in X \\ \hat{*}\ (X) & = & \mathbf{true} \end{array}$$

Informally speaking, the interpretation of arity is a predicate which puts a certain constraint on the "shape" of a set. For example, the first line tells us that the set X should contain exactly one element, etc.

Given a type $\varepsilon_1^{\alpha_1}\varepsilon_2^{\alpha_2}\alpha$ we define its valuation $\psi(\varepsilon_1^{\alpha_1}\varepsilon_2^{\alpha_2})$ as follows:

$$\psi\left(\epsilon_1^{\alpha_1}\epsilon_2^{\alpha_2}\right) = \{R \subseteq \psi(\epsilon_1) \times \psi(\epsilon_2) \mid \forall y \,.\, \hat{\alpha_1}\left(\pi_1^R(y)\right) \wedge \forall x \,.\, \hat{\alpha_2}\left(\pi_2^R(x)\right)\}$$

where

$$\begin{array}{rcl} \pi_1^R(y) & = & \{x \mid (x, y) \in R\} \\ \pi_2^R(x) & = & \{y \mid (x, y) \in R\} \end{array}$$

are first and second projections for a relation R. Informally, we interpret types as sets of all relations between interpretation of its endpoint entities which respect the constraints prescribed by endpoint arities.

Finally, we add an extra evaluation rule:

$$[\![\mathcal{R} : t_2]\!]\Delta = [\![\mathcal{R}]\!]\Delta \cap \psi(t_2)$$

Note, the application of this rule depends on the interpretation of types, but this is rather an expected phenomenon. Finally, with downcast defined, we have the complete syntax of our DSL (Fig. 5).

$$\begin{array}{rcll} \mathcal{R} & = & r & \text{— atomic relation} \\ & & \mathcal{R}^{-1} & \text{— inversion} \\ & & \mathcal{R} \circ \mathcal{R} & \text{— composition} \\ & & \mathcal{R} \cup \mathcal{R} & \text{— union} \\ & & \mathcal{R} \mid \mathcal{R} & \text{— choice,} \\ & & \mathbf{fix}\ [r \mapsto \mathcal{R}] & \text{— fixed point} \\ & & r : E, E & \text{— entity type downcast} \end{array}$$

*Fig. 5. The complete DSL syntax.*

Now we can state the soundness property of our type system.

**Theorem.** *Let*

$$\Gamma \vdash \mathcal{R} : t$$

*and let ψ be some entities valuation. Build the following evaluation environment Δ:*

$$\Delta = \{(r,\, R) \mid (r,\, t) \in \Gamma,\, R \in \psi(t)\}$$

*then*

$$[\![\mathcal{R}]\!]\Delta \in \psi\,(t)$$

In other words, if we can assign a type *t* to a certain relational expression in a certain typing environment then its evaluation will result in a relation which belongs to the set of relations, defined by the type *t*, under assumption that evaluation environment assigns all relevant atomic relations values, which are members of their types used in that typing environment. The proof of the theorem is by the induction on the typing derivation; it involves a careful case analysis of arity transformation but otherwise is simple.

## 4. *Implementation and Evaluation*

We implemented a prototype tool based on the ideas presented in the previous sections. Our tool consists of a DSL frontend, written in OCaml, and runtime support library, written in Java. The frontend parses a textual code model specification, performs type checking, and generates Java API for the model. This API makes use of the runtime support library to provide a convenient model-specific primitives for an end-user to manipulate the model. The runtime support library, written once and for all, is model-agnostic. It contains the implementation of generic binary relations and all DSL relational operators; thus, an end-user can transliterate relational expressions in the terms of the DSL directly in Java.

To evaluate our approach, we described a code model for a subset of Java and declaratively described the construction of control flow graph (CFG) from predefined atomic relations. We describe this evaluation here.

First, we give the entity-part of the model:

```
entity Expr;
entity Ref         : Expr;

entity Statement;
entity Assn        : Statement;
entity Conditional : Statement;
entity Loop        : Conditional;
entity If          : Conditional;
entity Seq         : Statement;
entity Return      : Statement;
entity For         : Loop;
entity WhileDo     : Loop;
entity DoWhile     : Loop;
```

We describe here abstractions for expressions and statements; we keep expressions mainly abstract since for our main example we do not need further detalisation; however, the actual specification contains the description of binary/unary expressions, variables, method calls, etc. For statements we build a conventional hierarchy of constructs.

Next, we provide basic atomic relations:

```
conditionOf : Expr, Conditional;
initOf      : Statement, For;
incrementOf : Statement, For;
thenOf      : Statement, If;
```

```
elseOf      : Statement?, If;
srcOf       : Expr, Assn;
dstOf       : Ref, Assn;
headOf      : Statement, Seq;
tailOf      : Statement?, Seq;
returnOf    : Expr?, Return;
```

In a nutshell, we reproduce here a representation of an abstract syntax tree (AST) with relations playing the role of AST's edges and entities playing the role of AST's nodes. Our next task is to define the CFG in terms of our model. As CFG, again, consists of nodes and edges, we use entities to represent nodes and relations to represent edges. First, we define the nodes of a CFG:

```
entity Node : Expr | Assn | Return;
```

This definition makes use of a construct we've never encountered before. But it actually does not add anything new: here we just declare a new entity "`Node`" to be a super-entity for "`Expr`", "`Assn`", and "`Return`". We use this extra construct since in the scenario we keep in mind a single model can be extended and customized in various ways thus it would be unreasonable to require a description to be build bottom-up since not all super-entities for an entity can be known in advance. Of course, a separate check is needed to ensure that the axioms of partial order are not violated.

Another question we have to discuss is why we define the nodes of a CFG in such manner. We argue that as the nodes we can takes the "primitive" statements which do not contain other statements; in our model those are just "`Assn`" and "`Return`". For conditional statements, however, the condition itself can be the node of a CFG – for example, the first piece of code of if statement, "visited" in the course of program execution, is its condition, which is an expression.

Our next task is to define a relation which associate each statement with its entry and exit nodes of a CFG. We do this by means of the following relations:

```
entryOf : Stmt, Node = fix [r ->
     (refl             : Assn, Assn) +
     (refl             : Return, Return) +
     (refl             : Expr, Expr) +
     (conditionOf'     : If, Node) +
     (conditionOf'     : WhileDo, Node) +
     (headOf' . r      : Seq, Node) +
     (bodyOf' . r      : DoWhile, Node) +
     (initOf' . r      : For, Node)
];

exitOf: Stmt, Node = fix [r ->
  (refl                    : Expr, Expr) +
  (refl                    : Return, Return) +
  (refl                    : Assn, Assn) +
  ((thenOf' + (elseOf' | conditionOf')) . r : If, Node) +
  ((tailOf' | headOf') . r : Seq, Node) +
  (conditionOf' . r        : WhileDo, Node) +
  (conditionOf'            : DoWhile, Node) +
  (conditionOf' . r        : For, Node)
];
```

The definitions are looking verbose (they, indeed, are), but in principle they are rather obvious. We would refrain from commenting on each line, but give the overall idea. First, both relations are recursive. Then, both are constructed by a case analysis: we construct a definition of the relation in question for a specific kind of statement and then union all of them. The definition for each kind of

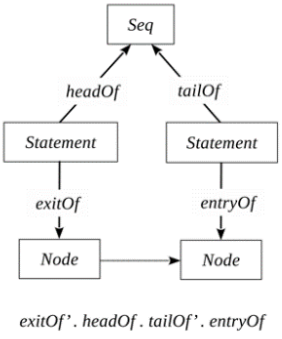
statement is rather straightforward: for example, if we wish to find an entry node for a "`Seq`" statement, we first need to take its first statement and then take that statement's entry node.

A relation "`headOf`" relates some statement to a "`Seq`" statement for which this statement is the head, thus "`headOf'`", the inversion, goes in the opposite direction and gives us the head; then we lookup for the entry node of the head via a recursive call (note "`. r`" part). Similarly, in the definition for the exit node for a "`Seq`" statement we need to find the "last" statement and retrieve its exit node via a recursive call. We already know than "`headOf'`" relation delivers us the head of the "`Seq`" statement; similarly, "`tailOf'`" delivers us the tail. However, in our model "`Seq`" statement may not have a tail; thus, we need to encode a pattern "if there is a tail, take the tail, otherwise take the head". This is exactly what "`tailOf'|headOf'`" expression does. Finally, primitive statements and expressions are their own entry/exit nodes. For this case we use a predefined constant "**`refl`**" which designates a reflexive relation which associate each entity to itself.

Finally, we need to specify the edges of the CFG. We do this via defining a relation which relates nodes to nodes:

```
edge: Node, Node =
exitOf'  . headOf . tailOf' . entryOf +
exitOf'  . (conditionOf : _, If) . (thenOf' + elseOf') . entryOf +
exitOf'  . (conditionOf : _, WhileDo) . bodyOf' . entryOf +
exitOf'  . (bodyOf : _, WhileDo) . conditionOf' . entryOf +
entryOf' . (conditionOf : _, DoWhile) . bodyOf' . exitOf +
entryOf' . (bodyOf : _, DoWhile) . conditionOf' . exitOf +
exitOf'  . (initOf : _, For) . conditionOf' . entryOf +
exitOf'  . (conditionOf : _, For) . bodyOf' . entryOf +
exitOf'  . (bodyOf : _, For) . incrementOf' . entryOf +
   exitOf'  . (incrementOf : _, For) . conditionOf' . EntryOf;
```

The definitions might look confusing, but actually are quite natural: we combine existing relations in order to relate exit node(s) of one statement with the entry node(s) of other(s). We give the intuition behind there expression using Fig. 6-8, each of which explains some group of lines in the definition given above. Bold arrows correspond to the derived relation for edges.



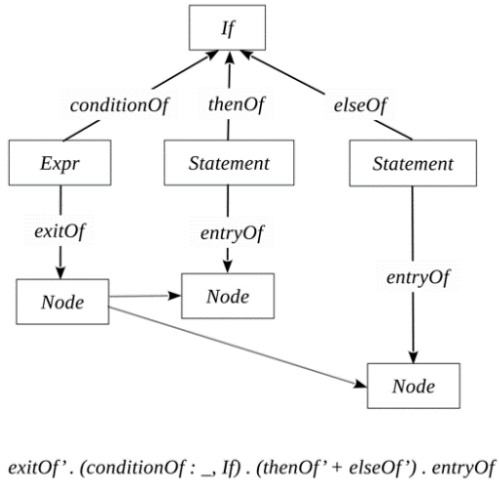
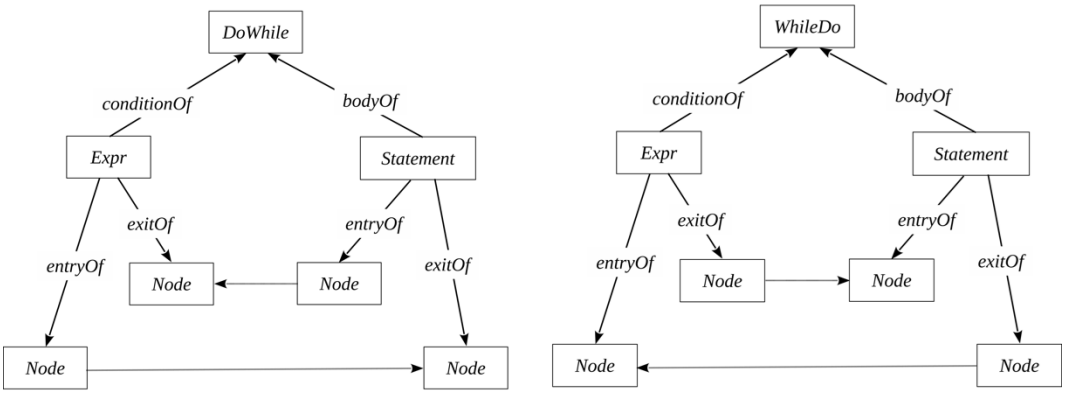




*Fig. 6. Edges for "Seq" and "If".*

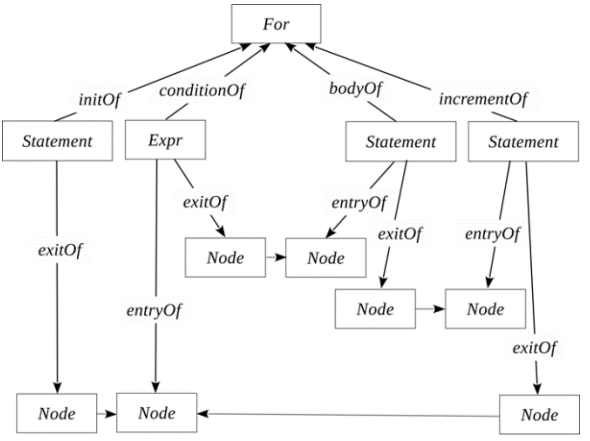
In order to evaluate the correctness of these descriptions we implemented a prototype of a client-side tool which parsers Java files and creates an instance of the model using the API generated by our frontend from model description. The results of constructed CFG and the source programs are shown in Fig. 9 and 10.




*entryOf'. (conditionOf : _, DoWhile) . bodyOf'. exitOf +*
*entryOf'. (bodyOf : _, DoWhile) . conditionOf'. exitOf*

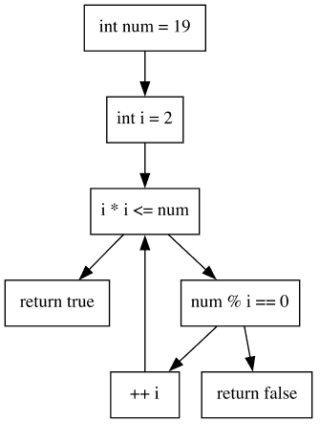
*exitOf'. (conditionOf : _, WhileDo) . bodyOf'. entryOf +*
*exitOf'. (bodyOf : _, WhileDo) . conditionOf'. entryOf*

*Fig. 7. Edges for "DoWhile" and "WhileDo".*




*exitOf'. (initOf : _, For) . conditionOf'. entryOf +*
*exitOf'. (conditionOf : _, For) . bodyOf'. entryOf +*
*exitOf'. (bodyOf : _, For) . incrementOf'. entryOf +*
*exitOf'. (incrementOf : _, For) . conditionOf'. entryOf*

*Fig. 8. Edges for the "For" loop.*




```
class IsPrime {
    public static boolean example() {
        int num = 19;
        for (int i = 2; i * i ⩽ num; ++i) {
            if (num % i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

*Fig. 9. Sample program and its CFG derived from the code model specification.*

```
class Conditionals {
    public static void main(String[] args) {
        int i = 3;
        i += 2;
        if (i > 5) {
            doA();
            doB();
        }
        while(i > 2) {
            int j = 0;
            while (j < 3) {
                i += j;
                ++j;
            }
        }
        if (i < 2) {
            doC();
        } else {
            doD();
        }
        System.out.println(i);
    }
}
```

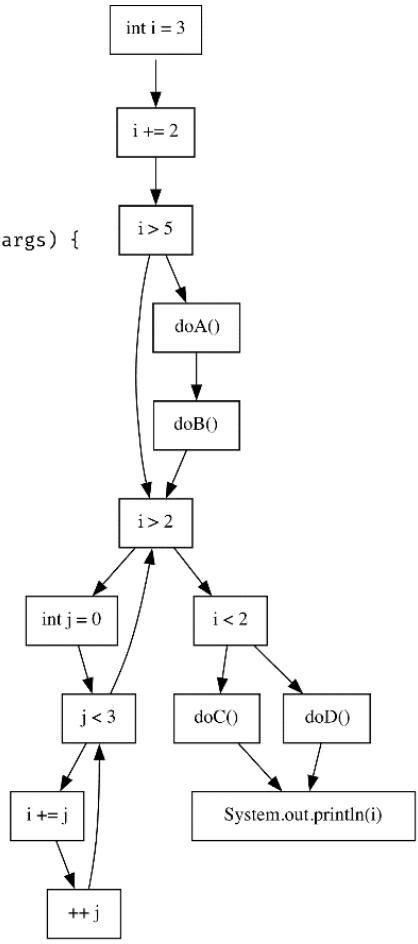



*Fig. 10. Sample program shows the handling of various control structures with nesting.*

## *4. Related Work*

Classical intermediate representations of programs such as abstract syntax tree, control flow graph or program dependence graph can be considered as the first instances of code models used in compilers and program analysis tools [2-3]. Abstract syntax tree represents the hierarchy of syntactic constructs. Besides compilation it can be used for example, for clone detection [4] or vulnerability analysis [5]. Control flow graph describes the control dependence between programs' statements; it is primarily used for control flow optimizations and data-flow analysis. Program dependence graph [6] was originally used to find the fragments of code which can influence the value of a certain variable. There are a lot of other specialized forms of program representations that are beneficient in various scenarios: static single assignment form (SSA) [7], three-address code, call graphs, etc. We position our approach not as a replacement for all these representations but rather as a way to declaratively describe the transitions from one representation to another and (potentially) as a system which would help to keep various representations synchronized with each other.

Another collection of approaches, which are in some aspects are very similar to ours and, in fact, serves as the motivation for our work, utilizes the notion of «code property graph». Property graph is one of widely used models of data representation in graph databases. It consists of a directed multigraph in which all nodes and edges are equipped with sets of attributes («properties», hence the name). Such graphs can be queried by traversals [7] which constitutes the basic way of their

utilization. Code property graph is a customization of this approach for program analysis where the attributes encode program-specific information [8]. This approach forms the basis for such tools as Joern [9] and CodeQL [10] which are used for vulnerability detection. A close approach is taken in [11] where an object-oriented query language is presented and evaluated for the task of computing code metrics. In comparison with our all these approaches advocate a language-agnostic solutions: the same structure is used to represent programs in various languages. We argue that our approach is can potentially provide a more robust results as it allows to maintain more code invariants. In addition, none of these approaches allows for defining new relations in a declarative form (although some queries may compensate for this deficiency).

## 5. Conclusion and Future Work

In this paper we presented an approach for describing language-specific relational code models. We described a model specification DSL, discussed its properties and shown some scenarios of its application. Our future work in this direction is to develop a more advanced implementation and perform a more thorough evaluation, including a quantitative one. There is also a number of research directions which are looking appealing.

First, an interesting question is the reusability aspect. While we are still going to stick with the language-oriented approach it is obvious that many languages share common properties. It would be interesting to investigate if it is possible to describe a generic meta-code-models from which a model for a concrete language could be built up with minimal customization.

Another interesting aspect is an efficient implementation of query language using advanced technics from the domain of graph or even relational databases.

Finally, as our language is strongly typed it is an interesting question if it is possible to use *type inhabitation* [1] approach in order to synthesize some useful relations from incomplete specifications.

## References

[1]. Benjamin C. Pierce. 2002. Types and Programming Languages (1st. ed.). The MIT Press. 648 p.

[2]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA.

[3]. Steven S. Muchnick. 1998. Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[4]. Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In Proceedings of the International Conference on Software Maintenance (ICSM '98). IEEE Computer Society, USA, 368.

[5]. Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12). Association for Computing Machinery, New York, NY, USA, 359–368. DOI: 10.1145/2420950.2421003.

[6]. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9, 3 (July 1987), 319–349. DOI: 10.1145/24039.24041.

[7]. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13, 4 (Oct. 1991), 451–490. DOI: 10.1145/115372.115320.

[8]. F. Yamaguchi, N. Golde, D. Arp and K. Rieck, Modeling and Discovering Vulnerabilities with Code Property Graphs, 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 2014, pp. 590-604. DOI: 10.1109/SP.2014.44.

[9]. Joern - The Bug Hunter's Workbench. Available at: https://github.com/joernio/joern, accessed 07.12.2025.

[10]. CodeQL Documentation. Available at: https://codeql.github.com/docs/, accessed 07.12.2025.

[11]. O. d. Moor et al., "Keynote Address: .QL for Source Code Analysis," Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), Paris, France, 2007, pp. 3-16. DOI: 10.1109/SCAM.2007.31.

## *Информация об авторах / Information about authors*

Дмитрий Евгеньевич ДУЖИНСКИЙ – программист, разработчик программных серверов прикладных систем.

Dmitry Evgenievich DUZHINSKII – computer programmer, backend developer.

Дмитрий Юрьевич БУЛЫЧЕВ – кандидат физико-математических наук, доцент кафедры системного программирования математико-механического факультета Санкт-Петербургского государственного университета. Область научных интересов – языки и инструменты программирования, функциональное, логическое и реляционное программирование, декларативное программирование, синтез программ.

Dmitry Yurievich BOULYTCHEV – Cand. Sci. (Phys.-Math,), associate professor at the Software Engineering Chair, faculty of Mathematics and Mechanics, Saint-Petersburg State University. Research interests: programming languages, compilers and tools, functional, logic and relational programming, declarative programming, program synthesis.