Programming and Debugging
with Semantically Lifted States

Eduard Kamburjan$^{(\boxtimes)}$, Vidar Norstein Klungre, Rudolf Schlatte,
Einar Broch Johnsen, and Martin Giese

Department of Informatics, University of Oslo, Oslo, Norway
{eduard,vidarkl,rudi,einarj,martingi}@ifi.uio.no

Abstract. We propose a novel integration of programming languages
with semantic technologies. We create a semantic reflection mechanism
by a direct mapping from program states to RDF knowledge graphs. This
mechanism enables several promising novel applications including the use
of semantic technology, including reasoning, for debugging and validat-
ing the sanity of program states, and integration with external knowl-
edge graphs. Additionally, by making the knowledge graph accessible
from the program, method implementations can refer to state semantics
rather than objects, establishing a deep integration between programs
and semantics. This allows the programmer to use domain knowledge
formalized as, e.g., an ontology directly in the program's control flow. We
formalize this integration by defining a core object based programming
language that incorporates these features. A prototypical interpreter is
available for download.

1 Introduction

Knowledge graphs and ontologies are eminently useful representations of formal
knowledge about individuals and universals. They are less suitable for the rep-
resentation of change, and in particular dynamic behavior. Different approaches
have been proposed that attempt to express program behavior in terms of actions
on a DL interpretation [26] or a DL knowledge base [1]. A recent approach has
combined a guarded command language with DL reasoning [6] to enable prob-
abilistic model checking over the combination. Each of these approaches entails
its own set of technical challenges, and they are also quite different from current
state-of-the-art programming paradigms.

In this work, we present a comparatively simple connection between programs
and knowledge graphs: we give a direct mapping of program states in an object-
based language to an RDF graph, including the running program's objects, fields,
and call stack. An immediate application of this mapping is to use semantic
technology for program debugging, visualisation, querying, validation, reasoning,
etc. In this paper, *semantic debugging* refers to the process of detecting and
correcting conceptual mistakes at the appropriate level of global object access
instead of interacting with only individual objects, as in conventional debuggers.

The RDF graph can be exposed within the programming language, which adds a semantic reflection layer to the programs. This enables *semantic programming* where the semantic view of the state can be exploited in the program. In particular, it allows one to use the knowledge of the application domain within the program. The programming language can further support a mechanism that extends the RDF graph with triples based on the results of specially designated methods. The approach is implemented and available.

In this work, the RDF graph is close to the object structure, thus representing an *internal domain* of the program. Our long-term goal is to use the same principles for *external domains* that correspond to a program's application domain. This will enable semantic integration between programs as well as linking them to external knowledge graphs and other data sources, thus extending ontology-based data access and integration (see e.g. [9]) to programs and behavioural specifications [13] that allow modeling of complex concurrent systems [16].

Contributions. The contributions of this paper are (1) the concept of interpreting a program state as a knowledge graph to enable semantic state access, (2) the application enabled by this concepts and (3) the SMOL language that implements the concept and exemplifies the applications.

Paper Overview. Section 2 gives a motivating example of mapping program states to an 'internal' knowledge graph. Section 3 gives syntax and semantics of SMOL, the used programming language, and defines the mapping of SMOL program states to RDF graphs. Section 4 extends SMOL with a statement to access its own knowledge graph, thereby adding reflection by semantic state access. Section 5 extends SMOL with a way to add *computed* triples to the knowledge graph. Section 6 describes the implementation, Sect. 7 discusses related work, and Sect. 8 concludes. Further details of this work may be found in an accompanying technical report [17].

2   Motivating Example

We start by giving an example of the use of domain models in programming to demonstrate semantic bugs and motivate semantic programming. Semantic bugs are programming errors that arise from mismatches between the implementation and the domain that is implemented. Such bugs are unlikely to cause immediate runtime errors and are, thus, harder to catch. Instead, they require to examine the program state through a conceptual lens, i.e., in terms of the implemented domain. *Semantic debugging* is the process of detecting and fixing such errors.

The implemented domain can be external or internal. An external domain relates the implementation to some concept outside the program. For example, a class modeling a car that contains a list of its wheels. If the list contains only one element, this is a semantic bug: the object does not represent a car. An internal domain relates the implementation to itself.

Example 1. Consider 2–3 trees [2], a data structure to access key-value pairs. Such trees have three kinds of nodes: leaves have one or two data values and no

children, 2-nodes that have one data value and two children and 3-nodes that
have two data values and three children. The keys are sorted within the tree; e.g.,
in a 2-node, all keys below the left child are smaller than the key stored in the
node and all below the right child are larger. Listing 1 shows an implementation
of such a tree. In the implementation, a node is a 2-node if the fields `dataR` and
`childR` are set to `null` and the other fields are non-null. A node is a 3-node if all
fields are non-null. A leaf has no children and at least one data value.

```
1 class Node(dataL, dataR, childL, childM, childR, parent)
2  get(k)
3   if(this.dataL = null)    then r := null; return r; end
4   if(k = this.dataL.key)   then r := this.dataL.value; return r; end
5   if(k <= this.dataL.key)  then r := this.childL.get(k); return r; end
6   if(this.dataR = null)    then r := this.childM.get(k); return r; end
7   ...
8  end
9 end
```

Listing 1. Part of the implementation of 2-3 trees in SMOL.

The nodes of a 2–3 tree change their "class" during their lifetime from leaf
to 2-node to 3-node. It would be highly inefficient to create a new object every
time this happens, as the addition of a value to the tree may cause several such
changes. This exemplifies how the application "domain" (here: 2–3 trees) and
the needs of the implementing language (here: efficiency) can collide.

In this example, the distinction between leaf nodes, 2-nodes and 3-nodes is
semantic in the sense that these nodes share the same syntactic structure. It
is not visible in the code that certain nodes are not supposed to exist; e.g.,
nodes where only `dataR` is set to `null` are so-called *faulty nodes*. This means that
erroneous insertion algorithms may cause *semantic bugs*: they may violate the
domain model of 2–3 trees. Note that faulty nodes as discussed above cannot be
described by means of an ontology for the syntax—the domain model is a model
of runtime states (i.e., the three kinds of nodes), not one of syntax.

An interpretation of the runtime state as a knowledge graph opens for *seman-
tic state access*; i.e., the runtime state can be queried by means of semantic tools
via this interpretation. In our example, all three kinds of node can be described
as both SHACL shapes and SPARQL queries for debugging.

Semantic bugs may cause *delayed* and *non-local* runtime errors. Semantic
debugging requires to perform semantic queries on a program state to access
the complete program state, not only the current stack trace. The faulty node
described above causes the tree to effectively ignore the values below `childR`
by the `get` method. Thus, the error is delayed (observable only after the faulty
insertion) and non-local (not observable with a single stack trace). Similarly, the
car with a single wheel may not cause *any* runtime error at all.

Semantic programming is the application of this conceptual lens from *within*
the program: As we interpret every runtime state as a model for a domain, we

can also perform queries automatically during execution, for example, querying for all faulty nodes and calling a repair function that fixes the tree.

Terminology. There are some conflicts of terminology between programming languages and semantic technologies, of which we emphasis the following: In programming languages the word *semantics* describes the runtime behavior of a program, and is unrelated to ontologies and formalized domains.

In programming, a *class* is completely described by (a) its name, (b) its fields and methods and (c) the name of its superclass. We only consider class hierarchies without multiple inheritance, which form a tree. Each *object* has an identifier and belongs explicitly to one class (and its superclasses). The state, i.e., the values in the fields of an object, may change, but the class does *not*. In contrast, an *OWL class* corresponds to a unary predicate in logic and any *resource* belongs to many classes. Classes are identified by their extension and not their name, and resources have no built-in notion of class membership.

3   Core Language `SMOL`

As our programming model, we consider a semantic minimal object language (`SMOL`). The language contains the minimal set of features to demonstrate semantic state access: a class system to define objects and a simple while language for statements. To demonstrate the use of reflection, `SMOL` uses dynamic typing: each value is tagged with its type at runtime.

3.1   Programming Model

A program in `SMOL` consists of a set of classes and a **main** block. Statements and expressions are standard, including a `null` reference and the self reference `this`. For simplicity all fields are public, fields are always prefixed with the target object and nested object creation and method calls inside expressions are not supported. The syntax of `SMOL` is given by the following definition.

Definition 1 (Surface Syntax). *Let v range over variables, f over fields, n over $\mathbb{N}$, m over method names and C over class names. The syntax of SMOL is defined below. We assume standard literals and operators in expressions. The notation $\overline{\cdot}$ denotes lists and $[\cdot]$ optional elements.*

Prog ::= $\overline{\text{Class}}$ **main** s **end**   Class ::= **class** C $(\overline{f})$ $\overline{\text{Met}}$ **end**   Met ::= $m(\overline{v})$ s **end**

  s ::= l:=e; | $\big[$l:=$\big]$se; | **if** e **then** s$\big[$**else** s$\big]$**end** s | s s | **while** e **do** s **end** s | **return** e; | **skip**

  se ::= **new** C$(\overline{e})$ | e.m$(\overline{e})$ e ::= `null` | l | n | e + e | e $\geq$ e | ...   l ::= `this`.f | e.f | v

The runtime semantics of `SMOL` is a transition system between runtime configurations. Each such configuration represents the state of the program at a given point of execution. An expression evaluates to a *domain element* (DE), which is either a literal value or an object reference. For method calls, we use runtime statements `rs` which extend statements `s` with a special statement $l \leftarrow$ `stack` (explained below). Runtime configurations are defined as follows.

Definition 2 (Configuration). *Let* X, Y *range over object identifiers,* σ, ς *over maps from variables to DEs,* ρ *over maps from fields to DEs and* i *over* $\mathbb{N}$. *Configurations* Conf, *objects* obs *and processes* prcs *are defined by the following:*

$$\begin{array}{ll}
\mathsf{Conf} ::= \mathsf{CT}\ \mathsf{obs}\ \big\langle[\mathsf{prcs}]\big\rangle & \mathsf{rs} ::= \mathsf{s}\ |\ \mathtt{l} \leftarrow \mathtt{stack;\ s} \\
\mathsf{obs} ::= (\mathtt{C}, \rho)_{\mathtt{X}}\ |\ \mathsf{obs\ obs} & \mathsf{prcs} ::= (\mathtt{m}, \mathtt{X}, \mathtt{rs}, \sigma)_i\ |\ \mathsf{prcs,\ prcs}
\end{array}$$

where CT *maps class names to a list of field names and a set of method entries. These are accessed by the auxiliary functions* fields$(\mathsf{CT}, \mathtt{C})$ *and* methods$(\mathsf{CT}, \mathtt{C})$, *respectively. We use* vars$(\mathsf{CT}, \mathtt{C}, \mathtt{m})$ *to access the variables and* body$(\mathsf{CT}, \mathtt{C}, \mathtt{m})$ *the body of a method. Terms* **obs** *are treated as sets.*

A runtime configuration Conf contains a set of objects and a stack of processes. An object has a unique name X and contains its class name C and memory ρ. A process has an id i and contains the name m of the method is executing, the object identifier X to resolve this, a runtime statement rs and a local store σ.

Observe that in a configuration, $\langle$prcs$\rangle$ realizes a *stack* of processes corresponding to nested method calls. In a process, the statement $\mathtt{l} \leftarrow \mathtt{stack}$ denotes that location $\mathtt{l}$ waits for a return value from the next process on the stack.

Definition 3 (Initial Configuration). *Given a program* Prog, *the initial configuration has the form* $\mathsf{CT}_{\mathsf{Prog}}\ (\mathtt{Entry}, \emptyset)_{\mathtt{E}}\ \big\langle(\mathtt{entry}, \mathtt{E}, \mathtt{s}, \emptyset)_1\big\rangle$, *where* $\mathsf{CT}_{\mathsf{Prog}}$ *is extracted as defined in Definition 2, but with an additional class* Entry *that has a single method* entry *with the statement of the main block as its body.*

The runtime semantics of SMOL is now presented as a structured operational semantics [24], i.e., a set of conditional rewrite rules which describe transitions from a runtime configuration into another. An expression evaluates to a pair X, f or a variable v if applied to a left-hand side and to a domain element if applied to a right-hand side. We denote by $[\![e]\!]_{Y}^{\sigma, \mathsf{obs}}$ the evaluation function for expressions e, where Y is the value of this, σ the local variables, and obs a set of objects (such that their memories ρ may be accessed). For brevity's sake, we refrain from introducing the full runtime semantics here and refer to our technical report [17].

Definition 4 (Transition System). *The most important rules of the transition system are given in Fig. 1.*

The transition system is defined with two layers. A global layer that performs a step of the whole system and local layer that performs a step of a single statement. To connect the two layers, rule **(lift)** performs a step in the top-most process, using a local transition relation $\xrightarrow{\mathsf{CT}}_{\mathtt{X}}$. The local transition relation considers only (a) the active statement (b) the current local memory and (c) all objects. We give three rules to illustrate this: **(af)** executes an assignment to some field. The left-hand side expression evaluates to the pair of object identifier and field name and the right-land side to a literal or reference to be stored

$$\text{(lift)} \ \frac{\mathtt{s}, \sigma, \mathsf{obs} \xrightarrow{\mathsf{CT}}_{\mathtt{X}} \mathtt{s''}, \sigma', \mathsf{obs}'}{\mathsf{CT} \ \mathsf{obs} \ \langle \mathsf{prcs}, (\mathtt{m}, \mathtt{X}, \mathtt{s\ s'}, \sigma)_i \rangle \to \mathsf{CT} \ \mathsf{obs}' \ \langle \mathsf{prcs}, (\mathtt{m}, \mathtt{X}, \mathtt{s''\ s'}, \sigma')_i \rangle}$$

$$\text{(af)} \ \frac{[\![\mathtt{e}]\!]_{\mathtt{Y}}^{\sigma, \mathsf{obs}\ (\mathtt{C}, \rho)_{\mathtt{X}}} = v \qquad [\![\mathtt{l}]\!]_{\mathtt{Y}}^{\sigma, \mathsf{obs}\ (\mathtt{C}, \rho)_{\mathtt{X}}} = \mathtt{X}, \mathtt{f}}{\mathtt{l} := \mathtt{e};, \sigma, \mathsf{obs} \ (\mathtt{C}, \rho)_{\mathtt{X}} \xrightarrow{\mathsf{CT}}_{\mathtt{X}} \mathbf{skip};, \sigma, \mathsf{obs} \ (\mathtt{C}, \rho[\mathtt{f} \mapsto v])_{\mathtt{X}}} \qquad \text{(av)} \ \frac{[\![\mathtt{e}]\!]_{\mathtt{Y}}^{\sigma, \mathsf{obs}} = v \qquad [\![\mathtt{l}]\!]_{\mathtt{Y}}^{\sigma, \mathsf{obs}} = \mathtt{v}}{\mathtt{l} := \mathtt{e};, \sigma, \mathsf{obs} \xrightarrow{\mathsf{CT}}_{\mathtt{X}} \mathbf{skip};, \sigma[\mathtt{v} \mapsto v], \mathsf{obs}}$$

$$\text{(new)} \ \frac{|\bar{\mathtt{e}}| = |\mathsf{fields}(\mathsf{CT}, \mathtt{C})| \qquad \mathtt{Y} \ \mathsf{fresh} \qquad \bigwedge_{1 \le i \le |\mathsf{fields}(\mathsf{CT}, \mathtt{C})|} \rho(\mathtt{f}_i) = [\![\mathtt{e}_i]\!]_{\mathtt{Y}}^{\sigma, \mathsf{obs}}}{\mathtt{l} := \mathbf{new} \ \mathtt{C}(\bar{\mathtt{e}});, \sigma, \mathsf{obs} \xrightarrow{\mathsf{CT}}_{\mathtt{X}} \mathtt{l} := \mathtt{Y};, \sigma, \mathsf{obs} \ (\mathtt{C}, \rho)_{\mathtt{Y}}}$$

Fig. 1. Selected rules of the transition system for SMOL.

there. The rule updates the heap of the target object and reduces the statement to **skip**. Rule **(av)** is similar if the left-hand side expression evaluates to a variable name. Rule **(new)** adds a new object after evaluating all parameter expressions.

Premises in the rules realize *dynamic checking*: mismatching parameters, null access and all other errors are caught at runtime. A runtime configuration for which no rule is applicable is *terminated*. A terminated runtime configuration with a non-empty stack is *stuck*. A program may get stuck if, e.g., a method is called on an object and this method is not defined in the object's class.

3.2   Semantic State Access

The formal definitions given in the previous section describe the global state of a program execution. The established way to examine this state in debuggers is to evaluate expressions in the top-most context and navigation of the process stack. To enable semantic state-access on the overall state without manual navigation, we map a configuration into an knowledge base using our SMOL domain model.

Definition 5 (Knowledge Base). *A knowledge base $\mathcal{K} = (\mathsf{T}, \mathsf{A})$ is a pair of a TBox T and an ABox A. We represent the ABox as a set $E \times P \times E$, where each element is a triple over entities E and predicates P. A triple (e_1, p, e_2) is also written $p(e_1, e_2)$.*

Entities are, e.g., domain elements or method names. We remind the reader that literal values are domain elements. The sets of predicates and entities may overlap: a field is used as both an entity (to express that a class has a field), as well as a property (to connect an object with the value stored within this field).

The TBox consists of axioms. Some axioms are generated as part of the mapping and stem from the SMOL domain. Additionally axioms to reflect the application domain can be provided by the user.

Definition 6 (SMOL Domain and Mapping). *The generic SMOL domain model without the subdomain for statements is the OWL model pictured in Fig. 2.*

The set of axioms defining the above model is denoted T_{SMOL}. Given a configuration Conf, the mapping $\mu(\mathsf{Conf})$ generates a ABox as defined in Fig. 3.
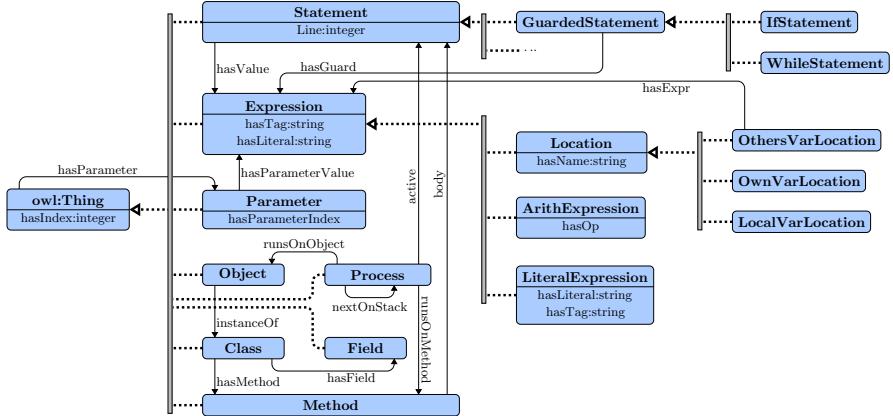
Fig. 2. OWL domain model for `SMOL` configurations. Common prefix `smol:` omitted. Boxes denote classes, arrows object properties and dotted arrows subclassing. Entries within the boxes are data properties.

$$\mu\big(\mathsf{CT}\ \mathsf{obs}\ \langle\mathsf{prcs}\rangle\big) = \mu(\mathsf{CT}) \cup \mu(\mathsf{obs}) \cup \mu(\langle\mathsf{prcs}\rangle)$$

$$\mu(\mathsf{CT}) = \bigcup_{\mathsf{C}\in\mathbf{dom}(\mathsf{CT})} \big\{\mathtt{smol:hasField}(f_\mathsf{P}(\mathsf{C}), f_\mathsf{P}(\mathsf{f})), \mathtt{rdfs:subProperty}(f_\mathsf{P}(\mathsf{f}), \mathtt{smol:Field}) \mid \mathsf{f} \in \mathsf{CT}(\mathsf{C})\big\}$$

$$\cup \bigcup_{\mathsf{C}\in\mathbf{dom}(\mathsf{CT})} \big\{\mathtt{smol:hasMethod}(f_\mathsf{P}(\mathsf{C}), f_\mathsf{P}(\mathsf{m})), \mathtt{rdf:type}(f_\mathsf{P}(\mathsf{m}), \mathtt{smol:Method}) \mid \mathsf{m} \in \mathsf{CT}(\mathsf{C})\big\}$$

$$\cup \big\{\mathtt{rdf:type}(f_\mathsf{P}(\mathsf{C}), \mathtt{smol:Class}) \mid \mathsf{C} \in \mathbf{dom}(\mathsf{CT})\big\}$$

$$\mu(\mathsf{obs}_1\ \mathsf{obs}_2) = \mu(\mathsf{obs}_1) \cup \mu(\mathsf{obs}_2) \qquad \mu\big((\mathsf{C}, \rho)_\mathtt{X}\big) = \big\{\mathtt{smol:instanceOf}(f_\mathsf{R}(\mathtt{X}), f_\mathsf{P}(\mathsf{C}))\big\} \cup \bigcup_{\mathsf{f}\in\mathbf{dom}(\rho)} \big\{\mathsf{f}(\mathtt{X}, \rho(\mathsf{f})\big\}$$

$$\mu(\langle\rangle) = \emptyset \qquad \mu(\langle\mathsf{p}_i\rangle) = \mu(\mathsf{p}_i) \qquad \mu(\langle\mathsf{p}_i\ \mathsf{p}_j\rangle) = \{\mathtt{smol:nextOnStack}(f_\mathsf{R}(i), f_\mathsf{R}(j))\} \cup \mu(\mathsf{p}_i) \cup \mu(\mathsf{p}_j)$$

$$\mu(\langle\mathsf{prcs}\ \mathsf{p}_i\ \mathsf{p}_j\rangle) = \{\mathtt{smol:nextOnStack}(i, j)\} \cup \mu(\langle\mathsf{prcs}\ \mathsf{p}_i\rangle) \cup \mu(\mathsf{p}_j)$$

$$\mu\big((\mathsf{m}, \mathtt{X}, \mathsf{rs}, \sigma)_i\big) =$$

$$\big\{\mathtt{smol:runsMethod}(f_\mathsf{R}(i), f_\mathsf{P}(\mathsf{m})), \mathtt{smol:runsOnObject}(f_\mathsf{R}(i), f_\mathsf{R}(\mathtt{X})), \mathtt{rdf:type}(f_\mathsf{R}(i), \mathtt{smol:Process})\big\}$$

$$\cup \big\{\mathtt{smol:active}(f_\mathsf{R}(i), \mu(\mathtt{rs}))\big\} \cup \bigcup_{\mathtt{v}\in\mathbf{dom}(\sigma)} \big\{f_\mathsf{P}(\mathtt{v})(f_\mathsf{R}(i), \sigma(\mathtt{v}))\big\}$$

Fig. 3. Mapping configurations. Statements and expression ommited for space reasons. f_P adds a prefix that identifies the program, f_R adds a prefix that identifies the run.

Fields and locations have two roles: All fields are properties (datatype or object properties, depending on the type), and they are elements of `Field`. Treating them as properties, we can use them to connect an object `O1` with an object `O2` stored in its field `f` using the triple `f(O1, O2)`. Treating them as individuals, we can express that they belong to a class. E.g., if `O1` is of class `C`, by using the triple `hasField(C, f)`. Analogously for variables and `Location`. The domain model also structures the statements, e.g., `IfStatement` and `WhileStatement` are both subclasses of `GuardedStatement`.[1]

---

[1] For space reasons, the complete version of this part of the domain model is given in the repository, and the following figures have been slightly simplified.

In the mapping, every class and object property is prefixed with an IRI unique to this version the program. We stress that each field is a subproperty of `Field` and every variable is a subproperty of `Location`. The `SMOL` domain is not merely a reformulation of the grammars in Definitions 1 and 2—it introduces terms like guarded statements which are not given in the grammar. Exactly one state is mapped to a knowledge base at a time, not a whole trace.

Example 2. Figure 4 shows a program and the main part of the mapping of its configuration. It exemplifies that the knowledge graph contains information about three layers of the program state: A syntactic layer describes the class information (in blue), an object layer describes the object instances (in brown) and a process layer describes the current stack (in yellow). The layers are not strict for fields and variables: `f` is both at the syntactic layer (`f` is a field of class `C`) and at the object layer (object `X` stores `2` in `f`) – the double line denotes equality.

```
1 class C(f) m(v) return this.f + v; end end
2 main o = new C(2); o.m(3); end
```
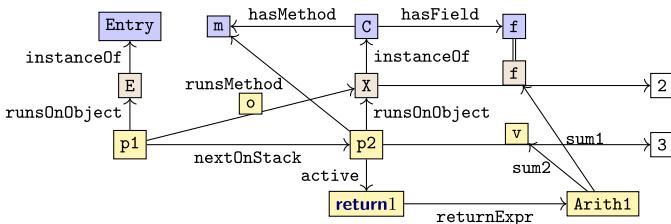

Fig. 4. A `SMOL` program and its mapping into a knowledge graph after `o.m(3)` is called.

The representation as a knowledge base allows us to access information beyond the basic notations of runtime semantics by inferring additional information through inference rules. We are generic in the access and inference mechanisms itself and assume some representation of axioms and queries.

Definition 7 (Queries). *An answering engine is a function that maps a knowledge base and a query to a set of answers:*

$$\mathsf{ans}\big((\mathsf{T},\mathsf{A}), q\big) = \big\{ \overline{x} \mid (\mathsf{T},\mathsf{A}) \models q(\overline{x}) \big\}$$

The satisfiability relation $\models$ depends on the concrete nature of the query.

Query engines can also handle boolean queries, such as description logic assertions, by returning either an empty set for true or a non-empty set for false.

Semantic Debugging. The mapping allows the programmer to debug a program by simple and efficient access to the runtime configuration through queries. This does not require an axiomatisation of the application domain, but if one is available, it can be used to additionally debug in terms of the application domain.

Example 3 (Semantic Debugging). Continuing with Example 1, we can formalize the notion of leaves, 2-nodes, 3-nodes and faulty nodes with the OWL class expressions from Fig. 5. To access a configuration Conf for semantic debugging with a query q, one needs to compute

$$\mathsf{ans}\big((\mathsf{T_{SMOL}} \cup \mathsf{T_{user}}), \mu(\mathsf{Conf})\big), q\big)$$

where $\mathsf{T_{user}}$ are user input axioms that define the domain of the application.

Note that the axioms $\mathsf{T_{SMOL}}$ and the mapping μ are part of the language and need *not* be provided by the user. The vocabulary is partially given by the language itself and partially by the user-defined classes. The overhead for the programmer is, thus, only to provide additional axioms if needed.

Recall that semantic state access is performed on a single state. For example, if the construction of the 2–3 tree in Example 1 temporarily contains a faulty node, then this node will not be retrieved by a semantic state access *after* the construction has completed (if the construction indeed fixes the node). If one wishes to detect such a faulty node without explicitly introducing it into the domain model, one can either use a query language with negation or a language for validation such as SHACL, whose integration into SMOL we discuss in Sect. 6.

Semantic debugging differs from traditional debugging in several points: (1) Access is performed with a query language to examine larger states, instead of manually investigating the call and heap structure of a snapshot. (2) The ability to query and debug using application domain knowledge. This becomes critical when investigating complex data structures where the implementation and the application domain conflict, as illustrated by Example 1 where the domain differentiates 3 classes of nodes, but the implementation has only one.

Semantic technologies are not a standard tool for debugging, but given the complexity of some current debugging tools, such as profilers, we expect that the overhead for the programmer is acceptable for complex structures, especially if they have an application domain component.

Root	$\equiv$ parent : null
Leaf	$\equiv$ (childL : null) $\sqcap$ (childM : null) $\sqcap$ (childR : null) $\sqcap$ ($\exists$dataL.$\exists$instanceOf.Pair)
TwoNode	$\equiv$ (childR : null) $\sqcap$ (dataR : null) $\sqcap$ ($\exists$childL.$\exists$instanceOf.Node)
	$\sqcap$ ($\exists$childM.$\exists$instanceOf.Node) $\sqcap$ ($\exists$dataL.$\exists$instanceOf.Pair)
ThreeNode	$\equiv$ ($\exists$dataL.$\exists$instanceOf.Pair) $\sqcap$ ($\exists$dataR.$\exists$instanceOf.Pair)
	$\sqcap$ ($\exists$childL.$\exists$instanceOf.Node) $\sqcap$ ($\exists$childM.$\exists$instanceOf.Node) $\sqcap$ ($\exists$childR.$\exists$instanceOf.Node)

Fig. 5. Domain model for 2–3 trees as OWL classes in DL syntax.

Verification. Before we investigate semantic state access further, we note that using established programming language constructs for control flow allows us to directly use results from programming languages. For example, we can carry over verification techniques and verify invariants. For a more detailed treatment of the following statement, we again refer to our technical report. We say that a

formula is *state-based* if it contains as predicates only (a) `instanceOf` and (b) subproperties of `Field` and `Expression`. An ontology is *state-based* if its axioms contain only such predicates.

Proposition. *Let* Prgm *be a* **SMOL** *program, such that type inference succeeds. Let O be a state-based domain ontology,* C *a class in* Prgm *and* φ *be a state-based description logic formula. There is a sound and complete (relative to integer arithmetic) proof system that checks whether* φ *is an invariant for* C.

4   Internal Semantic State Access

So far, semantic state access is used to query the program state from the outside to realize semantic debugging. Next, we integrate domain knowledge directly into the runtime semantics of the program: the control flow is expressed not in terms of data structure implementations, but in terms of the implemented domain. To do so, we add a new statement to retrieve information about the state through the conceptual lens of ontologies about the program state from the *inside*.

Definition 8 (Extended Surface Syntax). *Let str range over string literals. The syntax of* **SMOL**$^+$ *is defined by extending statements from Definition 1 as follows:*

$$\texttt{s ::= ... | l:=access}(str, \bar{\texttt{e}});$$

The runtime semantics of the `access` statement map the current program and use the *inferred* knowledge graph to perform some query. The query is not static: additional parameters are evaluated and mapped to nodes inside the knowledge graph. We use %i as placeholders in the query string to add the additional parameters. We require the existence of a `List` class to represent the results of the statement for further computations.

Definition 9. *The runtime semantics* **SMOL**$^+$ *are defined by all rules from Definition 3 and the following additional rule for the* `access` *statement. We remind that* I *and* R *are available as user inputs.*

$$(\textit{acc}) \frac{\texttt{q} = str\big[\%i \setminus [\![\texttt{e}_i]\!]_{\texttt{Y}}^{\sigma,\texttt{obs}}\big]_{i \leq n} \qquad l = \texttt{ans}\big(((\texttt{T}_{\texttt{SMOL}} \cup \texttt{T}_{\texttt{user}}), \mu(\texttt{Conf})), \texttt{q}\big)}{\underbrace{\texttt{CT obs} \langle \texttt{prcs}, \ (\texttt{m}, \texttt{Y}, \texttt{v := access}(str, \texttt{e}_1, \ldots, \texttt{e}_n); \ \texttt{s}, \sigma) \rangle}_{=\texttt{Conf}} \rightarrow_{\texttt{T}} \texttt{CT obs} \langle \texttt{prcs}, \ (\texttt{m}, \texttt{Y}, \texttt{s}, \sigma[\texttt{v} \mapsto l]) \rangle}$$

We illustrate internal SSA ith two examples: domain-specific control and reflection. We can apply the *domain* ontology in the queries, inspired by the 'ontology-mediated symbols' introduced by Dubslaff et al. [5], to directly control the program in terms of the domain. An ontology-mediated symbol is a query over the extended knowledge graph and can be expressed in our system.[2]

---

[2] Dubslaff et al. use description logic formulas for queries, not SPARQL, but our approach is general w.r.t. the query language.

Example 4. Consider the upper code in Listing 2. The scheduler uses the domain knowledge to determine which of the platforms is overloaded and accordingly moves servers between platforms. The critical point here is that `:Overloaded` is defined by the background knowledge and can, thus, be changed according to different scenarios outside the simulation.

```
1 class Platform(serverList) ... end   class Server(taskList) ... end
2 class Scheduler(platformList)
3   reschedule()
4     over := access("SELECT ?x WHERE{?x a :Overloaded }");
5     tasks := this.collectExcessiveTasks(over);
6     this.reschedule(tasks);
7   end
8 end
```

```
1 m(o)
2   callable :=
3     access("SELECT ?y WHERE{%1 :instanceOf ?y. ?y :hasMethod n }",o);
4   if callable <> null then o.n();   else .../*report error*/ end
5 end
```

Listing 2. Upper code: Using ontology-mediated symbols in `SMOL`. Lower code: Reflection with semantic state access.

Exchanging the background knowledge can be used to change the specification when a platform is overloaded—the language concepts of ontology-mediated programming are thus subsumed by semantic programming.

The class table and stack structure are both available in the knowledge graph. We can, thus, reason about these structures at runtime *in terms of the formal (runtime) semantics and domain.* I.e., this does not merely expose the structure of the implementing runtime environment but adds domain knowledge—in this case, the domain of runtime configurations. The actual implementation in the interpreter, or other runtime, may for efficiency be quite different.

Example 5. The method in the lower part of Listing 4 checks that a passed parameter is from a class that implements `n` before calling the method.

The above example illustrates a common pattern in languages with dynamic types or reflection and is based on the static information. As the knowledge graph underlying the state also enables access to the processes, one can also use it to make control based on the stack. E.g., one can bound recursion without an additional counter or reflect on the calling method without passing a parameter.

5   Computational Semantic State Access

So far, we can access the data in a configuration and, beyond merely serialising it into another format, use inference to ontologise it. However, we cannot access data that is implicit in the configuration. Consider Listing 3: The `Rectangle` class has a width `w` and a height `h`, but its area is not directly available. Even worse: each rectangle is part of a scene that may scale all its elements (and apply further

operations, which we omit here). While the final step of the computation of an area is a multiplication, the overall computation involves a method call.

```
1 class Scene(scaling) getScale() return this.scaling; end end
2 class Rectangle(scene, w, h)
3  rule area() s := scene.getScale(); return s*this.w*this.h; end
4 end
5 main sc := new Scene(2); ... r := new Rectangle(sc, 5, 1); end
```

Listing 3. A rectangle inside a scene.

Our solution is *computational semantic state access* (CSSA): certain methods, here `area` are directly encoded as inference rules to enrich the knowledge graph. This makes data that is a *computational* result available for inference. Furthermore, this allows one to determine based on a query where a computation has to be performed (instead of pre-performing it on all possible targets).

The language allows the programmer to mark methods as available for inference by exposing them with the `rule` keyword.

Definition 10 (Extended Surface Syntax with CSSA). *The syntax of Definition 8 is extended by replacing the method definition (from Definition 1) with*

$$\mathsf{Met} :: = [\texttt{rule}] \ \texttt{m}(\overline{\texttt{v}}) \ \texttt{s} \ \textbf{end}$$

For wellformedness, we demand that `m` is guaranteed to terminate if it is modified by `rule`.[3] The semantics of a `rule` method is not a transition rule, but an extension of the translation.

Definition 11. *The semantics of* SMOL^+ *is defined by the original transition system of* SMOL^+ *and by replacing the definition of* $\mu((\mathtt{C}, \rho)_\mathtt{X})$ *in Fig. 3 by*

$$\mu((\mathtt{C}, \rho)_\mathtt{X}) = \{\texttt{smol} : \texttt{instanceOf}(f_{\text{run}}(\mathtt{X}), f_{\text{prog}}(\mathtt{C}))\} \cup$$
$$\bigcup_{\mathtt{f} \in \mathbf{dom}(\rho)} \{f_{\text{prog}}(\mathtt{f})(f_{\text{run}}(\mathtt{X}), \rho(\mathtt{f}))\} \cup \bigcup_{\substack{\mathtt{m} \ is \ a \\ \texttt{rule} \ of \ \mathtt{C}}} \{f_{\text{prog}}(\texttt{exec\_C\_m})(f_{\text{run}}(\mathtt{X}), l)\}$$

where l *is a literal computed as follows. Let* $\mathcal{K}$ *be a knowledge graph and* $\mu_{obj}^{-1}(\mathcal{G}) = (\mathsf{CT} \ \textsf{obs} \ \epsilon)$ *its state without the processes. Let* $\mathtt{X}$ *be the object id of the object bound to* $?o$. *The configuration* $\mathsf{CT} \ \textsf{obs} \ \langle(\mathtt{m}, \mathtt{X}, \mathtt{s}, \{\})_1\rangle$ *finishes in a configuration of the form* $\mathsf{CT} \ \textsf{obs}' \ \langle(\mathtt{m}, \mathtt{X}, \textbf{return} \ \mathtt{e}, \sigma)_1\rangle$. *The literal* l *is defined as the evaluation of* $\mathtt{e}$ *in this configuration. If the execution does not finish in a configuration of the required form, we set* $l = \texttt{smol:null}$.

The execution is not performed on $\mu(\mathsf{Conf})$ itself. This means that any state change, e.g., object creations or changes of fields, are *not* recorded in $\mu(\mathsf{Conf})$. We stress that the `access` statement is still part of the language.

---

[3] Via a timeout, a syntactic check that no (mutual) recursion and no loops occur during its execution, or a termination proof. We do not commit to a concrete restriction.

Example 6. Consider the final configuration of the program in Listing 3. Let X be the created object. The method `area` is syntactically guaranteed to terminate and application to X results in the following added triple:

$$\texttt{prog:exec\_Rectangle\_area}(\texttt{run}:\texttt{X}, 10)$$

We remind the reader that this is not merely an arithmetic expression, but requires a method call to include the scaling of the overall scene—`rule`-methods allow one to include such computations directly into the knowledge graph.

A method exposed with `rule` takes no parameters to avoid spurious triples. If a call with a particular parameter is required, one may introduce a wrapper class that is created before $\mu(\mathsf{Conf})$ is computed.

Example 7. To lookup 5 in Example 1 in a query, one may introduce the following class and create an instance `new Wrap(tttree, 5)` before a `get` call.

```
1 t.get(key); return v; end end
```

CSSA is not redundant to method calls: it extends the knowledge base at *every* instance and allows to select in the graph depending on these attributes.

6   Implementation

An implementation is available at `github.com/Edkamb/SemanticObjects`. It supports a superset of SMOL$^+$ and adds inheritance and some convenience features for the program, such as output. Several examples are provided, including the example from Sect. 2 with complete, and a SMOL$^+$ implementation of a subset of the geological assistant [4], a simulator for geological processes.

Interpreter. SMOL$^+$ is implemented by an interpreter written in Kotlin that builds on Apache Jena and HermiT [10] for the semantic state access. The interpreter implements an interactive shell to realize a Read-Evaluate-Print-Loop (REPL) that allows the user to step through the execution and semantically access the current state. The user may query the state with SPARQL, validate it against SHACL shapes or retrieve all members of an OWL class defined by a class expression. As a domain ontology, a file containing OWL classes can be loaded. It is possible to run a program without stepping through the execution; the language is extended with a `breakpoint` statement that stops the execution in this case. Additionally, the interpreter uses the prefixes `prog:` (for f_P) and `run:` (for f_R) to simplify referencing elements of the current state. CSSA is implemented by introducing a Jena-functor for each `rule` method and a rule with this functor in its head. The functor copies the complete interpreter state without the stack and executes the corresponding method.

Performance. We have evaluated the performance of internal SSA by adding n elements to a 2–3 tree and the querying OWL class expression using HermiT. For $n = 100$, the system used 150 s. To evaluate CSSA, we have similarly

added n elements and then used the `rule` wrapper from Example 7 to retrieve a value. Here, for $n = 3000$, the system used 205 s. 3000 added values correspond to approximately 9000 created `SMOL` objects[4] and 60k triples. This shows that our proof-of-concept can handle non-trivial amounts of data, and complex data structures and queries. We conjecture that the most significant bottleneck concerning the performance is the non-optimised interpreter, and the explicit generation of the knowledge base. To increase efficiency, we plan to make the knowledge base virtual and only access object states as required to answer queries. Backward reasoning can be included e.g. by query rewriting for OWL QL ontologies.

7   Related Work

Ontologies for Java's core concepts by Kouneli et al. [18] and for connecting object-oriented languages by de Aguiar et al. [3] have similarities to `SMOL`'s ontology, but aim at communication between users and not at semantic state access.

Imperative programming languages and transition systems can operate directly on knowledge graphs through atomic actions. Golog [22] uses first-order logic guards to examine and pick elements from its own state. *knowledge-based* programs [8] support an epistemic knowledge modality K. Zarrieß [26] integrates description logic in a concurrent extension of Golog to verify CTL properties with description logic assertions. These assertions are easily realised using `assert` in `SMOL`$^+$, while our object invariants are orthogonal to CTL checking of traces. When operating on a knowledge graph, an ABox may change and violate a TBox. Calvanese et al. [1] propose two operations `ASK` and `TELL` for transition systems defined *explicitly* over knowledge bases. `ASK` corresponds roughly to our `access`, while `TELL` performs an action required by the explicit representation. In contrast, the transition system in `SMOL`$^+$ is *implicit* such that well-established principles from programming languages carry over to avoid reinvestigations of modularity, runtime semantic structure and control flow for knowledge bases. While all changes to the knowledge graph are global in Calvanese et al., global changes in `SMOL`$^+$ only happen in the part of the knowledge graph inferred from user-provided axioms; the part inferred from the mapping only changes locally.

Our work has not investigated programming languages that operate directly on DL interpretations or knowledge graphs, as done by [1,26], but rather how programming languages can be enhanced by semantic technologies. Closest to our work, *ontology-mediated* programming [5,6] defines an interface to integrate additional knowledge into a stochastic model checking tool, using external knowledge graphs to influence control flow. In contrast, we use internal knowledge graphsfor debugging. Neither the application to debugging nor an integration with rule-based inference as in CSSA has been studied in any of these approaches. Additionally, these approaches all use unconventional operators or highly specialised paradigms, while `SMOL`$^+$ allows external semantic state access for a standard object-oriented language; the `access` and `rule` extensions are optional

---

[4] Partly because the implementation still lacks a garbage collector for rewritten nodes.

and based on a clear interface and established query language, instead of low-level logic-based operations. For these reasons, $\mathtt{SMOL}^+$ appears as conceptually simpler.

Ontologies can be used to type programs. Leinberger et al. [19] study DL concept expressions as static types in a λ-calculus, and type check using SHACL constraints [21]. Existing programming languages can be integrated with RDF data using the type systems of Paar and Vrandecic [23] and Leinberger et al. [20]. The difference between ontologies and regular types is not merely one of taste: (a) concepts allow more expressive structure than type hierarchies and (b) classes in programming languages are designed by the user to fit the needs of its application, while the concepts of the domain are designed to accomodate the needs of a general domain. While this work attempts to unify two tools made for different tasks, our approach is to give a sensible interface. $\mathtt{SMOL}$ is dynamically typed and the concepts of the domain and mapping in $\mathtt{SMOL}$ are disjoint and need to be connected using additional axioms. The connection to types has also been investigated through mappings [15] and code generation [25].

Eiter et al. [7] explore answer set programming to embed rules over DL knowledge graphs in the declarative setting of logic programs. Their rules are more expressive than our CSSA rules and aim to be a general programming approach. Käfer and Harth [14] perform actions on RDF files in the semantic web using linked data, operating on a set of user-input rules for an abstract state machine. Horne et al. define an operational semantics for SPARQL updates [12] and a system that internalizes queries into a process algebra [11].

8   Conclusion

This paper presents a novel approach to combine semantic technologies and programming languages. By regarding runtime configurations as knowledge graphs, we can use semantic state access to query such configurations for semantic debugging. By adding a semantic reflection layer to the programming language, computations can be driven by the result of queries from within a program. Finally, a deep integration of inference and computation allows inference to trigger method executions through computational semantic state access.

Future Work. As discussed, we plan to make the knowledge base virtual for performance reasons. We are also considering to develop an extension with special statements to manipulate the TBox and investigate how further programming languages concepts, such as garbage collection and encapsulation, carry over.

References

1. Calvanese, D., Giacomo, G.D., et al.: Actions and programs over description logic knowledge bases: a functional approach. In: Knowing, Reasoning, and Acting: Essays in Honour of Hector J. Levesque. College Press (2011)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press (2001)
3. de Aguiar, C.Z., de Almeida Falbo, R., Souza, V.E.S.: OOC-O: a reference ontology on object-oriented code. In: Laender, A.H.F., Pernici, B., Lim, E.-P., de Oliveira, J.P.M. (eds.) ER 2019. LNCS, vol. 11788, pp. 13–27. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33223-5_3
4. Din, C.C., Karlsen, L.H., Pene, I., Stahl, O., Yu, I.C., Østerlie, T.: Geological multi-scenario reasoning. In: Proceedings of Norsk Informatikkonferanse (NIK) (2019)
5. Dubslaff, C., Koopmann, P., Turhan, A.-Y.: Ontology-mediated probabilistic model checking. In: Ahrendt, W., Tapia Tarifa, S.L. (eds.) IFM 2019. LNCS, vol. 11918, pp. 194–211. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34968-4_11
6. Dubslaff, C., Koopmann, P., Turhan, A.: Give inconsistency a chance: semantics for ontology-mediated verification. In: Description Logics, Volume 2663 of CEUR Workshop Proceedings. CEUR-WS.org (2020)
7. Eiter, T., et al.: Combining answer set programming with description logics for the semantic web. Artif. Intell. **172**(12–13), 1495–1539 (2008)
8. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Knowledge-based programs. Distrib. Comput. **10**, 4 (1997)
9. Giese, M., et al.: Optique: zooming in on big data. IEEE Comput. **48**(3), 60–67 (2015)
10. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: HermiT: an OWL 2 reasoner. J. Autom. Reason. **53**(3), 245–269 (2014). https://doi.org/10.1007/s10817-014-9305-1
11. Horne, R., Sassone, V.: A verified algebra for linked data. In: FOCLASA, Volume 58 of EPTCS (2011)
12. Horne, R., Sassone, V., Gibbins, N.: Operational semantics for SPARQL update. In: Pan, J.Z., et al. (eds.) JIST 2011. LNCS, vol. 7185, pp. 242–257. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29923-0_16
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
14. Käfer, T., Harth, A.: Rule-based programming of user agents for linked data. In: LDOW@WWW, Volume 2073 of CEUR. CEUR-WS.org (2018)
15. Kalyanpur, A., Pastor, D.J., Battle, S., Padget, J.A.: Automatic mapping of OWL ontologies into Java. In: SEKE (2004)
16. Kamburjan, E., Hähnle, R., Schön, S.: Formal modeling and analysis of railway operations with active objects. Sci. Comput. Program. **166**, 167–193 (2018)
17. Kamburjan, E., Klungre, V.N., Schlatte, R., Johnsen, E.B., Giese, M.: Programming and debugging with semantically lifted states (full paper). Research report 499, Department of Informatics, University of Oslo, March 2021. https://ebjohnsen.org/publication/rr499.pdf

18. Kouneli, A., Solomou, G., Pierrakeas, C., Kameas, A.: Modeling the knowledge domain of the Java programming language as an ontology. In: Popescu, E., Li, Q., Klamma, R., Leung, H., Specht, M. (eds.) ICWL 2012. LNCS, vol. 7558, pp. 152–159. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33642-3_16

19. Leinberger, M., Lämmel, R., Staab, S.: The essence of functional programming on semantic data. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 750–776. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_28

20. Leinberger, M., Scheglmann, S., Lämmel, R., Staab, S., Thimm, M., Viegas, E.: Semantic web application development with LITEQ. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8797, pp. 212–227. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11915-1_14

21. Leinberger, M., Seifer, P., Schon, C., Lämmel, R., Staab, S.: Type checking program code using SHACL. In: Ghidini, C., et al. (eds.) ISWC 2019. LNCS, vol. 11778, pp. 399–417. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30793-6_23

22. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: GOLOG: a logic programming language for dynamic domains. J. Log. Program. **31**(1–3), 59–83 (1997)

23. Paar, A., Vrandečić, D.: Zhi# – OWL aware compilation. ESWC 2011. LNCS, vol. 6644, pp. 315–329. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21064-8_22

24. Plotkin, G.: A structural approach to operational semantics. J. Log. Algebr. Program. **60**, 17–139 (2004)

25. Stevenson, G., Dobson, S.: Sapphire: generating Java runtime artefacts from OWL ontologies. In: Salinesi, C., Pastor, O. (eds.) CAiSE 2011. LNBIP, vol. 83, pp. 425–436. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22056-2_46

26. Zarrieß, B., Claßen, J.: Verification of knowledge-based programs over description logic actions. In: IJCAI. AAAI Press (2015)