GEV: Statically Correct and Programmable Knowledge Graph Updates

Eduard Kamburjan[1,2] , **Shqiponja Ahmetaj**[3] , **Chinmayi Prabhu Baramashetru**[4] , **Paolo Pareti**[5]
[1]IT University of Copenhagen, Denmark
[2]University of Oslo, Norway
[3]TU Wien, Austria
[4]University of Kent, United Kingdom
[5]University of Southampton, United Kingdom
eduard.kamburjan@itu.dk

Abstract

Knowledge Graphs (KGs) evolve over time and it is critical to ensure that their integrity constraints are maintained after each update. We introduce GEV, the first tool to statically ensure that a KG update in Java preserves satisfaction of SHACL constraints. This allows verification of updates at design time, and eliminates the need for costly continuous revalidation. GEV is a command-line system that loads and verifies updates, applies them to a loaded KG, and keeps track of the validation status. Internally, it relies on *SHACL graph updates*, a theoretical framework with a method for static verification.

1 Introduction

The Knowledge Graphs (KGs) data model has many applications in data engineering, data integration and ontological modeling [Hogan *et al.*, 2022]. Unlike relational data, graphs have no fixed schema and are instead constrained by defining required *shapes* that nodes and edges must conform to. SHACL (SHApes Constraint Language) is the W3C-recommended standard for expressing such constraints in RDF KGs, with validators and related tools evolving rapidly [Ke *et al.*, 2024; Seifer *et al.*, 2024; Ahmetaj *et al.*, 2025a].

Applications require that KGs satisfy these shapes even as they evolve. However, ensuring that graph modifications preserve SHACL validation remains a significant challenge. Graph updates are frequent [Polleres *et al.*, 2023], and the state of the art relies on (partial) revalidation after each change [Zacouris *et al.*, 2025], an expensive process that cannot be performed at design time. Recent work introduced a framework for statically verifying whether certain graph updates preserve SHACL shape satisfaction independently of a concrete graph [Ahmetaj *et al.*, 2025b]. However, this framework is purely theoretical and relies on a *declarative* update language, whereas in practice graphs are modified using *operational* programming libraries.

This paper introduces GEV, the first tool that statically verifies Java-based KG modifications against SHACL constraints. GEV enables developers to safely manage graph evolution by: (1) writing graph modifications in standard Java using the Apache Jena API, (2) statically verifying at design time that the resulting graph updates preserve SHACL satisfaction, and (3) applying verified updates repeatedly *without the need for revalidation.* The user provides an initial KG, a set of SHACL shapes, and a set of graph modifications as Java methods using Apache Jena. GEV is a command line interface to manage the KG. It extracts graph updates from these methods, validates the initial graph once against the shapes, and verifies that each update preserves the shapes. Once verified, an update can be safely applied to any graph that satisfies the SHACL shapes, guaranteeing that the graph remains valid after the update. Figure 1 shows an example workflow for GEV in practice, where first a graph (line 1) and shapes are loaded (line 2) and used for validation (line 5). The loaded modifications are verified against them (line 9). Afterwards, the modifications are applied for a concrete value (line 11).

Our main contribution is a system description of the first tool to statically verify Java programs w.r.t. KG modifications. It is modular and extendable, with new modifications being expressed in standard Java code using the popular Apache Jena library: updates are provided as compiled Java methods and loaded at runtime. Going beyond prior approaches [Kamburjan and Kostylev, 2021; Leinberger *et al.*, 2019], GEV is the first tool to realize statically correct KG updates using mainstream technologies (SHACL, RDF, Java).[1]

2 Background

We provide brief background on KGs, SHACL, and the SHACL-based graph update language underlying our approach. We refer to [Ahmetaj *et al.*, 2025b] for full formal definitions and theoretical results.

Knowledge Graphs and SHACL. We consider RDF KGs, viewed as finite sets of triples of the form (a, p, b) or $(a, \text{rdf:type}, B)$, where B is a class name, p a property, and a, b are nodes.

SHACL *shape graphs* consist of a set of *shapes* describing constraints, and a set of *targets* specifying the nodes they apply to. E.g., the following shape states that every patient must

[1]Video: https://drive.google.com/file/d/12HmbxtvEKKYX7vh_tKIcuDE5KgjxQQl7
GitHub: https://github.com/Edkamb/GraphUpdateInterface

GEV

```
GEV> load graph graph_rdf.ttl
GEV> load shapes shapes_rdf.ttl
GEV> list shapes
shape-patient: Not validated
GEV> list updates
add-patient: No verification
GEV> validate shape-patient
success
GEV> verify add-patient shape-patient
success
GEV> apply add-patient ex:patient1
shape-patient: Validated
```

Figure 1: Example use of GEV for knowledge graph management.

be either active or discharged:

$$\texttt{PatientShape} \leftrightarrow \textit{ActivePatient} \vee \textit{DischargePatient}.$$

Together with the target $(\textit{Patient}, \texttt{PatientShape})$, this specification requires that every *Patient* instance satisfies the shape PatientShape. A graph *validates* a SHACL specification if all targeted nodes satisfy their associated shapes.

SHACL-based Graph Updates and Static Validation. Knowledge graphs evolve over time through updates that add or remove facts. To reason about such evolution in the presence of SHACL shapes graphs, we use a declarative graph update language that is tightly integrated with SHACL. Updates are expressed as sequences of elementary actions that add or remove class assertions or property edges, where affected nodes can be selected using SHACL shape expressions.

For example, the following update discharges a patient p:

$$\begin{aligned}&\texttt{REMOVE}(p, \text{rdf:type}, \textit{ActivePatient})\\ &\texttt{ADD}(p, \text{rdf:type}, \textit{DischargePatient})\\ &\texttt{REMOVE}(*, \textit{treatsPatient}, p)\end{aligned}$$

Intuitively, this update removes the information that p is active, marks p as discharged, and removes all treatment relations involving p. Updates can be parameterized, composed into sequences, and guarded by SHACL-based preconditions.

The central reasoning problem supported by this framework is *static validation under updates*: given a SHACL specification S and an update U, determine whether for every graph G that validates S, the updated graph $U(G)$ also validates S. An update U is said to *preserve* S if this condition holds. This notion of preservation enables reasoning about the correctness of graph modifications independently of concrete data. To decide static validation under updates, a regression technique is used that incorporates the effects of update actions directly into the SHACL constraints. This reduces update preservation to an (un)satisfiability check of a derived SHACL specification, forming the theoretical basis for the static verification approach realized by GEV.

3 System Description

GEV consists of three main components: a *modification engine*, an *extraction component*, and a *verification engine*. The main goal of GEV is to provide an interface to evolve KGs in a way that (a) uses a mainstream language and library to describe modification and (b) minimizes the number of validations. In particular, revalidation after applying an update is not necessary if the update has been verified against the SHACL shapes. Technically, GEV infers all possible sequences of operations on the KG from the provided graph updates, translates these sequences in graph updates and then uses the existing verification engine for graph updates to verify that the updates preserve validity of the modified graph w.r.t. the provided SHACL shapes. Consequently, applying the graph modification also preserves it.

Figure 3 shows an example update, which takes a node for an active patient, sets it to discharged, and removes all edges that express that they are currently treated. Note that the update is parametric – it can be reused many times in graph evolution. The graph update that is extracted from it is below, and indeed abstracts from Java specifics, such as the used library, or the auxiliary features such as string operations. The last element of the figure is a SHACL shape that is expresses that every patient is active if and only if the patient is not discharged. This shape is preserved by this modification.

Additionally, GEV also provides an interface to keep track of verification and validation results Whenever the graph is evaluated against a SHACL shape, this information is saved. When a graph modification is applied which has been shown to preserve certain shapes, then only the validation information for all other shapes is removed. In the following, we investigate the main components in more detail.

Graph Update Extraction. GEV verifies Java methods using the Apache Jena API. These methods must be provided in a compiled form as a `.jar` file and have the following signature, for any number n of resources.

```
static void M(Model m, Resource r, ..)
```

We use the static analysis framework SootUp [Karakaya *et al.*, 2024] to extract all possible sequences of Jena API calls on the parameter `model`. Currently, GEV support addition and removal of triples via `Model.add` and `Model.remove`, which are abstracted into ADD and REMOVE graph operations. However, using Java gives us the possibility for flexible programming of graph updates. The static analysis supports constant folding for URLs, auxiliary methods for code reuse, branching, local variables and constants, all concepts needed for flexible programming which are missing from the abstract graph update framework.

If the method uses any other API call on the parameter `model`, then no update is extracted. Otherwise, they are passed to the verification engine.

Graph Update Verification Verification is performed using the SHACL2FOL tool [Pareti, 2024], which converts SHACL shape graphs into equisatisfiable FOL theorems, that is, theorems that are satisfiable if and only if there exists a graph that satisfies the SHACL shapes. This tool represents theorems in the TPTP [Sutcliffe and Suttner, 1998] format, and uses the Vampire theorem prover [Kovács and Voronkov, 2013] to determine satisfiability or containment [Pareti *et al.*, 2022]. This satisfiability check is used to verify static validation under updates, after applying the regression technique directly into the FOL translation of SHACL shapes.

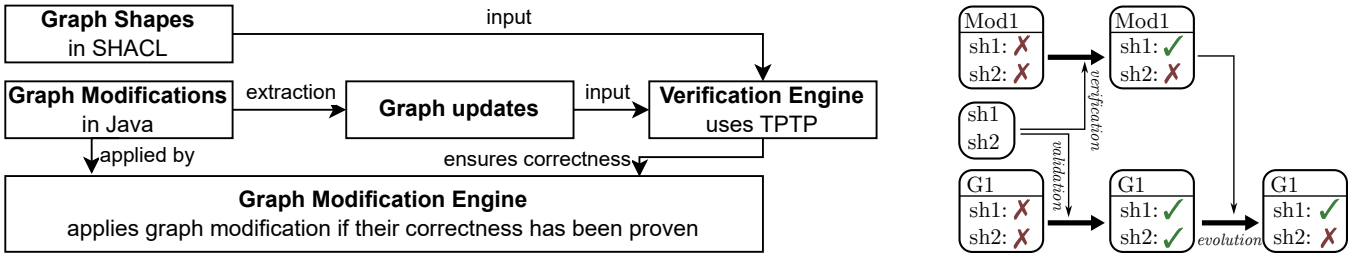

Figure 2: Overview over the internal structure of GEV (left) and an example workflow (right). The modification Mod1 is verified to preserve shape sh1. The graph G1 is first validated against sh1 and sh2, and upon applying Mod1 it is not necessary to revalidate sh1.

Java

```
static public void add-Patient(Model model, Resource patient) {
  String prefix = "https://smolang.org/health/";
  Resource active = model.createResource(prefix+"ActivePatient");
  Resource discharge = model.createResource(prefix+"DischargePatient");
  Property treats = model.createProperty(prefix+"treatsPatient");
  model.remove(patient, RDF.type, activePatient);
  model.add(patient, RDF.type, dischargePatient);
  model.removeAll(null, treatsPatient, patient); } //null acts as a wildcard
```

Updates

```
//for any node @p1, removes this triple
REMOVE(@p1, rdf:type, h:ActivePatient),
//for any node @p1, adds this triple
ADD(@p1, rdf:type, h:DischargePatient),
//removes the below triples for any subject
REMOVE(*, h:treatsPatient, @p1)
```

SHACL

```
:shape-patient a sh:NodeShape;
 sh:targetClass :Patient;
 sh:xor (
  [sh:class :ActivePatient]
  [sh:class :DischargePatient]
) .
```

Figure 3: Overview over internal structure.

Graph Modification Engine The graph modification engine keeps track of loaded graphs, shapes and modifications. It also keeps track of which shapes each graph is currently satisfying and which shapes each modification is shown to preserve. Lastly, it applies modifications to the underlying graph. Figure 2 shows an example on the right. The example contains one graph modification (Mod1), one graph (G1), and two SHACL shapes (sh1, sh2). First, the graph is validated against the two shapes (1), before the update is verified against them (2). In this example, the modification only preserves shape sh1. The last step (3) is to apply the modification to the graph. The modification engine marks that validity of sh1 is preserved, and only sh2 requires revalidation.

4 Usage

In interactive mode, GEV operates with a command line interface, where each command can be issues manually by the user. This is intended for exploration and DataOps, where predefined modifications and shapes have to be applied to an existing graph using novel data or shapes. The most important commands are shown in table 1 and illustrated in fig. 1. In scripting mode, GEV takes a sequence of commands and executes them in order. It automatically aborts if one of the following cases occurs: (1) Graph validation against a shape fails, or (2) verification of a graph modification fails. For more complex conditions or logic, GEV has a Java API.

Command	Description
`load graph`	Loads an RDF graph
`load shapes`	Loads SHACL shapes
`list updates`	Lists updates and their status
`list shapes`	Lists shapes and their status
`apply shape`	Validates the graph against a shape
`apply update`	Applies a graph update on the graph
`verify update`	Verifies an update against a shape
`write graph`	Writes an RDF graph to disk

Table 1: Overview over internal structure.

While we have to forgo a full performance evaluation here for brevity's sake, we can report that already on small graphs and the example of fig. 3 with the original shapes from Ahmetaj *et al.* [2025b], the performance gain is substantial: For a synthetic graph with 500000 triples, of which 80000 are `treatsPatient` edges. Validating the shapes takes ∼3 seconds, while verifying the modification takes ∼0.5 seconds, and applying it takes ∼0.2 seconds. Thus, already for small graphs our approach outperforms revalidation by a factor of 12. An in-depth performance evaluation remains future work, but we note that as the graph size increases, its validation time also increases, but *verification time* remains constant (w.r.t. fixed SHACL shapes) and application is a very cheap operation, compared to validation.

Acknowledgements

This work is partially supported by the EU project *SM4RTENANCE* (101123490), the DFF project *Graph-based Verification of Reflective Programs* (5254-00016B) and the Horizon Europe projects DataPACT (101189771) and RAISE Suite (101188337). Ahmetaj was supported by the Austrian Science Fund (FWF) and netidee SCIENCE [T1349-N], and the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201].

References

Shqiponja Ahmetaj, Iovka Boneva, Jan Hidders, Katja Hose, Maxime Jakubowski, José Emilio Labra Gayo, Wim Martens, Fabio Mogavero, Filip Murlak, Cem Okulmus, Axel Polleres, Ognjen Savkovic, Mantas Simkus, and Dominik Tomaszuk. Common foundations for shacl, shex, and pg-schema. In *Proceedings of the ACM on Web Conference 2025, WWW 2025, Sydney, NSW, Australia, 28 April 2025- 2 May 2025*, pages 8–21. ACM, 2025.

Shqiponja Ahmetaj, George Konstantinidis, Magdalena Ortiz, Paolo Pareti, and Mantas Simkus. SHACL validation under graph updates. In *ISWC (1)*, volume 16140 of *Lecture Notes in Computer Science*, pages 140–157. Springer, 2025.

Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. *ACM Comput. Surv.*, 54(4):71:1–71:37, 2022.

Eduard Kamburjan and Egor V. Kostylev. Type checking semantically lifted programs via query containment under entailment regimes. In *Description Logics*, volume 2954 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021.

Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. Sootup: A redesign of the soot static analysis framework. In *TACAS (1)*, volume 14570 of *Lecture Notes in Computer Science*, pages 229–247. Springer, 2024.

Jin Ke, Zenon G. Zacouris, and Maribel Acosta. Efficient validation of SHACL shapes with reasoning. *Proc. VLDB Endow.*, 17(11):3589–3601, 2024.

Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

Martin Leinberger, Philipp Seifer, Claudia Schon, Ralf Lämmel, and Steffen Staab. Type checking program code using SHACL. In *ISWC (1)*, volume 11778 of *Lecture Notes in Computer Science*, pages 399–417. Springer, 2019.

Paolo Pareti, George Konstantinidis, and Fabio Mogavero. Satisfiability and containment of recursive shacl. *Journal of Web Semantics*, page 100721, 2022.

Paolo Pareti. SHACL2FOL: an FOL toolkit for SHACL decision problems. *CoRR*, abs/2406.08018, 2024.

Axel Polleres, Romana Pernisch, Angela Bonifati, Daniele Dell'Aglio, Daniil Dobriy, Stefania Dumbrava, Lorena Etcheverry, Nicolas Ferranti, Katja Hose, Ernesto Jiménez-Ruiz, Matteo Lissandrini, Ansgar Scherp, Riccardo Tommasini, and Johannes Wachs. How does knowledge evolve in open knowledge graphs? *TGDK*, 1(1):11:1–11:59, 2023.

Philipp Seifer, Daniel Hernández, Ralf Lämmel, and Steffen Staab. From shapes to shapes: Inferring SHACL shapes for results of SPARQL CONSTRUCT queries. In *Proceedings of the ACM on Web Conference 2024, WWW 2024*, pages 2064–2074. ACM, 2024.

Geoff Sutcliffe and Christian Suttner. The tptp problem library. *Journal of Automated Reasoning*, 21:177–203, 1998.

Zenon Zacouris, Jin Ke, and Maribel Acosta. UpSHACL: Targeted Constraint Validation for Updates over Knowledge Graphs. In *International Semantic Web Conference*, pages 122–139. Springer, 2025.