

Constraint-Guided RDF Construction with Provenance

Xuemin Duan¹[0000–0002–3256–8341], David Chaves-Fraga²[0000–0003–3236–2789],
and Anastasia Dimou¹[0000–0003–2138–7972]

¹ KU Leuven – Flanders Make@KULeuven – Leuven.AI, Leuven, Belgium
{xuemin.duan,anastasia.dimou}@kuleuven.be

² CiTIUS, Universidade de Santiago de Compostela, Spain
david.chaves@usc.es

Abstract. Validating RDF graphs against constraint languages, such as SHACL, remains decoupled from the RDF graph’s construction, usually performed via mapping languages, such as RML. This leads to blindly generating all triples defined by the mapping, including those irrelevant to or non-conforming with the SHACL shapes, incurring construction and validation overhead. Avoiding unnecessary overhead requires a precise characterization of the triples to be generated, at three progressively finer facets: those that are shape-relevant, those that belong to non-conforming or conforming focus nodes, and those that constitute minimal deterministic evidence. While such a characterization is implicit in a provenance semantics for SHACL over materialized RDF graphs, it has not been exploited during the graph’s construction. Instead of deriving these characterizations a posteriori, we propose compiling SHACL shapes into an existing RML mapping following the provenance semantics, producing rewritten mappings for each above facet. We prove correctness of the approach, implement it as an open-source system, experimentally show up to three orders of magnitude faster execution and less memory than the post-hoc baseline. By lifting constraints into the RDF graph construction, our approach reduces the need for post-hoc validation over the full graph, avoiding the materialization of the entire graph including triples that are irrelevant or non-conforming. Our work provides the foundation for seamlessly integrating the construction and validation of RDF graphs into a coherent unified process, reducing redundancy, promoting consistency by design and improving overall efficiency.

Keywords: SHACL · RML · RDF Validation · Provenance

1 Introduction

Validating RDF graphs, using declarative constraint languages (e.g., SHACL [19]), remains decoupled from graph construction, using declarative mapping languages (e.g., RML [12,16]), and is applied only after the full RDF graph has been materialized. All graph triples are not only “blindly” generated independent of their relevance or conformance to the SHACL shapes, but they also incur unnecessary construction and validation overhead [1,18,24].

SHACL, beyond a validation language [19], is now widely adopted to define application profiles [4,28], implicitly characterizing the triples relevant for downstream applications as those that validation against the shapes would inspect. Yet, all derivable triples are typically constructed nowadays regardless of relevance. While mapping pruning driven by SPARQL queries has been recently explored [24], no such approach exists based on SHACL. Among relevant triples, non-conforming data can compromise application reliability, yet post-hoc validation poses severe scalability challenges [1,18,24]. Moreover, even if only non-conforming triples were identified and selectively generated, retaining all of them does not explain precisely why a violation holds. This requires identifying the minimal deterministic evidence, i.e., the minimal set of triples necessary for the result to hold [5].

Determining which triples of a constructed RDF graph are relevant and contribute as evidence to its non-conformance or conformance result to a given SHACL shapes graph is a why-not- and why-provenance problem. Such a provenance semantics for SHACL has been formalized over materialized graphs [10], implicitly characterizing the triples at three progressively finer facets: relevance, conformance, and minimal deterministic evidence. However, no existing work lifts constraints into mappings to couple construction and validation.

We propose compiling SHACL shapes into existing RML mappings, building on the provenance semantics for SHACL [10], producing rewritten mappings that realize each facet during construction, while achieving why-not provenance through shape negation. We prove correctness of the approach, inheriting sufficiency and conformance from the provenance semantics. This work mainly involves RML-Core and RML-FNML [16], RML Views [2,23], R2RML [8], and SHACL Core [19]. We implement the approach as an open-source SHACL2RML¹, experimentally confirming correctness and showing up to three orders of magnitude faster execution and less memory than the post-hoc baseline [10].

We contribute the first formalization and implementation of mapping rewriting that supports: (1) selective generation of only shape-relevant triples; (2) exclusive generation of the triples belonging to non-conforming or conforming focus nodes; and (3) generation of minimal deterministic evidence for non-conformance or conformance. We further (4) classify and discuss SHACL constraints by their compilability into RML, identifying the supported fragment and the remaining constraints as directions for future extension. Our approach reduces construction and validation overhead by surfacing (non-)conformance already during construction. It provides the foundation for seamlessly integrating the construction and validation of RDF graphs into a coherent unified process, reducing redundancy, promoting consistency by design, and improving efficiency.

The remainder of this paper is structured as follows. Section 2 establishes the formal background on SHACL, data provenance for SHACL, and RML. Section 3 classifies constraints, defines the approach, and proves its correctness. Section 4 describes the implementation and experimental validation. Section 5 introduces related work. Section 6 concludes with limitations and future work.

¹ <https://github.com/dtai-kg/SHACL2RML>

2 Preliminaries

This section establishes the formal background on RDF, SHACL, data provenance for SHACL [10], and RML mappings adopted throughout this paper. Remarks 1–5 state our assumptions on SHACL and RML constructs.

2.1 RDF and SHACL

Definition 1 (RDF). Let $\mathcal{I}$, $\mathcal{L}$, $\mathcal{B}$ be pairwise disjoint infinite sets of *IRIs*, *literals*, and *blank nodes*, and let $\mathcal{N} = \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$. An *RDF graph* G is a finite set of triples $(s, p, o) \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times \mathcal{N}$. We define an equivalence relation $\sim$ on $\mathcal{L}$ for language tag equality and a strict partial order $<$ on $\mathcal{L}$ for value comparison.

Definition 2 (SHACL Schema and Shape Definition). A *schema* $\mathcal{H}$ is a finite set of *shape definitions* (s, φ, τ) with a shape name $s \in \mathcal{I} \cup \mathcal{B}$, a shape expression φ (Definition 4), and a target declaration τ (Definition 3). A *focus node* of (s, φ, τ) is a node v satisfying τ . An RDF graph G *conforms* to $\mathcal{H}$, written $G \models \mathcal{H}$, iff $\mathcal{H}, G, v \models \varphi$ for every $(s, \varphi, \tau) \in \mathcal{H}$ and every focus node v of (s, φ, τ) , where conformance $\mathcal{H}, G, a \models \varphi$ is defined inductively over the shape expression grammar following [10], reproduced in Appendix A.1 for reference.

Definition 3 (Target Declaration [17]). In this work, a *target declaration* τ takes one of the following shape expression forms: $\geq_1 \text{rdf:type.hasValue}(c)$ for `sh:targetClass` c [17], $\geq_1 p.\top$ for `sh:targetSubjectsOf` p , and $\geq_1 p^-. \top$ for `sh:targetObjectsOf` p , all monotone; a shape without a target takes $\tau = \neg \top$.

Remark 1. `sh:targetNode` is excluded as it designates a specific IRI that cannot be statically aligned to mapping rules.

Remark 2. We assume $\mathcal{H}$ is non-recursive, and restrict `sh:path` to direct predicates $p \in \mathcal{I}$, leaving composite path expressions, including the `rdfs:subClassOf` closure of class targets, beyond this work.

Definition 4 (Shape Expression [10,17]). A shape expression φ is given by:

$$\begin{aligned} \varphi ::= & \top \mid \perp \mid \text{hasShape}(s) \mid \text{hasValue}(c) \mid \text{test}(t) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg \varphi \mid \#_n^m p.\varphi \mid \forall p.\varphi \\ & \mid \text{eq}(p, p') \mid \text{disj}(p, p') \mid \text{closed}(P) \mid \text{lessThan}(p, p') \mid \text{lessThanEq}(p, p') \mid \text{uniqueLang}(p) \end{aligned}$$

where s is a shape name, $c \in \mathcal{N}$, $p, p' \in \mathcal{I}$, $m, n \in \mathbb{N}$, $P \subseteq \mathcal{I}$. We write $\geq_n p.\varphi$ when $m = \infty$ and $\leq_m p.\varphi$ when $n = 0$, and use $\#_n^m p.\varphi$ for the general case. The correspondence between SHACL Core constructs and the formal notation follows [10,17]. In particular, `sh:minCount` n and `sh:maxCount` m are expressed as $\#_n^m p.\top$, `sh:qualifiedValueShape` φ' with `sh:qualifiedMinCount` n' and `sh:qualifiedMaxCount` m' as $\#_{n'}^{m'} p.\varphi'$, and remaining property-level constraints, except `sh:hasValue`, as $\forall p.\varphi'$. We abstract node-level tests as elements of a set Ω , partitioned into *type tests* $\Omega_T = \{\text{nodeKind}(k), \text{datatype}(d), \text{langIn}(L)\}$ and *value tests* $\Omega_V = \{\text{in}(S), \text{minIncl}(v), \text{maxIncl}(v), \text{minExcl}(v), \text{maxExcl}(v), \text{minLen}(l), \text{maxLen}(l), \text{pattern}(r)\}$, where $k, d \in \mathcal{I}$, $L \subseteq \mathcal{L}$, $S \subseteq \mathcal{N}$, $v, r \in \mathcal{L}$, and $l \in \mathbb{N}$. All shape expressions are assumed to be in *negation normal form* (NNF) with negation pushed inward to atomic expressions.

Example 1. Given triples $(:p1, a, :Article)$, $(:p1, :year, "2028")$, $(:p1, :title, "A")$, $(:p2, a, :Article)$, $(:p2, a, :JournalArticle)$, $(:p2, :year, "2026")$, $(:p2, :title, "B")$, and $(:p2, :journal, :J1)$ in G , consider a shape definition $(:ArticleShape, \varphi_1, \tau_1)$, where $\tau_1 = \geq_1 \text{rdf:type.hasValue}(:Article)$ and $\varphi_1 = \forall :year. \text{test}(\text{maxIncl}(2026))$, and a shape definition $(:JournalShape, \varphi_2, \tau_2)$, where $\tau_2 = \geq_1 \text{rdf:type.hasValue}(:JournalArticle)$ and $\varphi_2 = \forall :journal. \text{test}(\text{nodeKind}(\text{IRI})) \wedge \forall :year. \text{test}(\text{datatype}(\text{xsd:integer}))$. Focus nodes of $:ArticleShape$ are $:p1$ and $:p2$. $:p1$ is non-conforming as $2028 > 2026$, while $:p2$ conforms, as does its focus node for $:JournalShape$.

2.2 Data Provenance for SHACL

The provenance semantics of SHACL [10] formalizes, for any node v , shape expression φ , and RDF graph G , the *neighborhood* $B(v, G, \varphi)$ as the subgraph of G consisting of those triples that show that v conforms to φ ; if v does not conform, the neighborhood is empty. By applying the neighborhood to the negation $\neg\varphi$, the same mechanism identifies the triples explaining non-conformance. The *shape fragment* collects the neighborhoods of all conforming focus nodes across all shape definitions in a schema. The neighborhood is designed to be deterministic and sufficient, and as minimal as possible [10]; the full inductive neighborhood definition is given in [10], reproduced in Appendix A.2.

Intuitively, the neighborhood of a conforming node v in G contains only triples whose predicates are in $P(\varphi)$ (first facet), the set of predicates appearing in φ (e.g., p, p' in Definition 4), extended for $\neg\text{closed}(P')$ with all predicates not in P' . Conformance of a node follows the SHACL specification [19] and fully defined in Appendix A.1 (second facet). For a conforming node, the neighborhood retains all p -triples for $\forall p.\varphi'$, as every object node must satisfy φ' , but only those p -triples whose object nodes satisfy φ' for $\geq_n p.\varphi'$, and only those whose object nodes do not satisfy φ' for $\leq_n p.\varphi'$. $\text{eq}(p, p')$ contributes all triples via both p and p' ; $\text{disj}(p, p')$, $\text{lessThan}(p, p')$, $\text{lessThanEq}(p, p')$, $\text{closed}(P')$, and $\text{uniqueLang}(p)$ contribute an empty neighborhood, while their negated forms contribute triples satisfying the negated constraint (third facet).

Definition 5 (Neighborhood and Shape Fragment [10]). The *neighborhood* of a node v in graph G with respect to φ in the context of a schema $\mathcal{H}$, written $B(v, G, \varphi)$, is defined by induction on φ (Appendix A.2). If $\mathcal{H}, G, v \models \varphi$ then $B(v, G, \varphi) = \emptyset$. The *shape fragment* of G with respect to schema $\mathcal{H}$ is:

$$\text{Frag}(G, \mathcal{H}) = \bigcup \{B(v, G, \varphi \wedge \tau) \mid (s, \varphi, \tau) \in \mathcal{H}, v \in \mathcal{N}\}$$

Theorem 1 (Sufficiency [10, Theorem 3.4]). If $\mathcal{H}, G, v \models \varphi$, then $\mathcal{H}, G', v \models \varphi$ for any RDF graph G' with $B(v, G, \varphi) \subseteq G' \subseteq G$.

Theorem 2 (Conformance [10, Theorem 4.1]). If $\mathcal{H}$ has monotone targets and $G \models \mathcal{H}$, then $\text{Frag}(G, \mathcal{H}) \models \mathcal{H}$.

Example 2. Continuing Example 1, we have $P(\varphi_1) = \{:\text{year}\}$ and $P(\varphi_2) = \{:\text{year}, :\text{journal}\}$. $B(:p1, G, \varphi_1 \wedge \tau_1)$ is empty, since $:p1$ does not conform ($2028 > 2026$). $B(:p2, G, \varphi_1 \wedge \tau_1) = \{(:p2, :year, "2026"), (:p2, a, :Article)\}$. The shape fragment $\text{Frag}(G, \mathcal{H})$ additionally includes $B(:p2, G, \varphi_2 \wedge \tau_2)$.

2.3 RML Mappings

We formalize RML following the RML specification [12,16]. An RML mapping Σ consists of a finite set of *triples maps*, each specifying how to generate RDF triples via one *logical source*, one *subject map*, and zero or more *predicate-object maps*, from individual source records. RML follows a *record-at-a-time* processing model: each triples map is evaluated independently for each record in its logical source, with no access to other records during evaluation [16].

Definition 6 (Logical Source and Record). A *record* $r : \text{Ref} \rightarrow \mathcal{N}$ is a partial function from reference names to RDF terms, where $r(rf) = \text{none}$ denotes a null value. A *logical source* ls takes one of the following forms:

$$ls ::= \text{rmlBase}(src, rf, it) \mid \text{rdbBase}(tbl) \mid \text{rdbView}(q) \mid \text{rmlView}(q')$$

corresponding to file-based source with locator $src \in \mathcal{I} \cup \mathcal{L}$, reference formulation $rf \in \mathcal{I}$, and optional iterator $it \in \mathcal{L} \cup \{\text{none}\}$ [12]; relational table $tbl \in \mathcal{L}$ [8,14]; SQL query $q \in \mathcal{L}$ over relational databases [8,14]; RML views of [2,23], adopted here, with SQL query $q' \in \mathcal{L}$ over file-based sources.

Definition 7 (Term Map). A *term map* generates an RDF term from a record with an optional *term type* $tt \in \{\text{IRI}, \text{Literal}, \text{BlankNode}\}$, defined as:

$$tmap ::= \text{const}(c, tt) \mid \text{ref}(rf, tt) \mid \text{tmpl}(T, \bar{rf}, tt) \mid \text{fexec}(fe, \rho, tt)$$

The evaluation $\text{eval}(tmap, r)$ of a term map on a record r yields a term in $\mathcal{N}$, or none if any referenced value is absent [8,12]. A reference may in general yield multiple values within one record, while our work is limited to the single-valued case (Remark 3). $\text{const}(c, tt)$ always evaluates to c ; $\text{ref}(rf, tt)$ to the value of rf in r ; $\text{tmpl}(T, \bar{rf}, tt)$ to the value obtained by substituting each $rf_i \in \bar{rf}$ into template T ; and $\text{fexec}(fe, \rho, tt)$ to the output of executing fe selected via return map ρ , itself a term map, which may be omitted and abbreviated as $\text{fexec}(fe)$.

Remark 3. We further assume eval is single-valued per record, as RML engines split a multi-valued record into several records, on each of which an injected function can only be evaluated separately.

Definition 8 (Function Execution). A *function execution* [16] $fe = (f, \text{Inp})$ pairs an FNO function IRI $f \in \mathcal{I}$ with a finite set of inputs Inp , each a pair $(par, tmap)$ of a parameter IRI and a term map supplying its value. Since $tmap$ may itself be fexec , executions may be nested. The evaluation is defined as: $\text{eval}(\text{fexec}(fe, \rho, tt), r) = f\left(\left\{par \mapsto \text{eval}(tmap, r) \mid (par, tmap) \in \text{Inp}\right\}\right)[\text{eval}(\rho, r)]$.

A *subject map* sm is a term map. A *predicate-object map* $pom = (pm, om)$ pairs a *predicate map* pm and an *object map* om . An object map is a pair $(tmap, lit)$ where $lit \in \{\text{dt}(d), \text{lang}(\ell), \text{none}\}$ specifies an optional datatype or language tag, or a *referencing object map* $\varrho = (id_p, jcs)$ joining with a parent triples map via join conditions, whose evaluation follows [16]. Named graph declarations are preserved unchanged on each pom during rewriting and omitted from the formalization.

Remark 4. We assume each subject map is injective over its logical source, i.e., distinct records generate distinct subjects; for IRI templates this can be ensured via safe separators [30]. Predicate maps are restricted to $\text{const}(p)$ with $p \in \mathcal{I}$.

Definition 9 (Triples Map and RML Mapping). A *triples map* $tm = (id, ls, sm, poms)$. We write $sm(tm)$, $poms(tm)$, $ls(tm)$ for the respective components, and $poms_p(tm) = \{(pm, om) \in poms(tm) \mid pm = \text{const}(p)\}$ for predicate-object maps with predicate p . Any `rr:class c` declaration is encoded as a *pom*, abbreviated as $(\text{const}(a), \text{const}(c))$. The RDF triples generated by tm are:

$$\llbracket tm \rrbracket = \{ (s, p, o) \mid r_c \in ls, (pm, om) \in poms, s = \text{eval}(sm, r_c), p = \text{eval}(pm, r_c), o = \text{eval}(om, r_c), s, p, o \neq \text{none} \}$$

An *RML mapping* Σ is a finite set of triples maps and $\llbracket \Sigma \rrbracket = \bigcup_{tm \in \Sigma} \llbracket tm \rrbracket$.

Remark 5. We assume every tm in Σ whose sm can generate instances of a class c in $\llbracket \Sigma \rrbracket$ declares a *pom* $(\text{const}(a), \text{const}(c))$, as the triples maps generating the same focus node cannot otherwise all be identified and aligned.

Example 3. Given $\Sigma = \{tm_{art}, tm_{jour}\}$ that produces G from Example 1.

$$\begin{aligned} sm_{art} &= \text{tmpl}(\text{http://article.org/\{id\}_\{title\}}, [\text{id}, \text{title}]) \\ pom_1 &= (\text{const}(a), \text{const}(\text{Article})) \quad pom_2 = (\text{const}(\text{:title}), (\text{ref}(\text{title}), \text{dt}(\text{xsd:string}))) \\ pom_3 &= (\text{const}(\text{:year}), (\text{ref}(\text{year}), \text{dt}(\text{xsd:integer}))) \\ sm_{jour} &= \text{tmpl}(\text{http://article.org/\{doi\}_\{title\}}, [\text{doi}, \text{title}]) \\ pom_4 &= (\text{const}(a), \text{const}(\text{Article})) \quad pom_5 = (\text{const}(a), \text{const}(\text{:JournalArticle})) \\ pom_6 &= (\text{const}(\text{:year}), (\text{ref}(\text{year}), \text{dt}(\text{xsd:integer}))) \\ pom_7 &= (\text{const}(\text{:journal}), \text{ref}(\text{journal}, \text{IRI})) \end{aligned}$$

Both triples maps produce the same subject IRI as `:p2`, so that no single triples map governs the complete *neighborhood* of the focus node `:p2` for both shapes.

3 Our Approach

This section presents our approach to compiling a SHACL schema into an RML mapping. Given a schema $\mathcal{H}$ and a mapping Σ , the goal is to produce rewritten mappings Σ'_{RE} , Σ'_{CO} , and Σ'_{EV} at three progressively finer facets, relevance, conformance, and evidence, under the assumptions of Remarks 1–5 and the supported constructs of Table 1 in Section 3.1.

We define the *relevant triples* of a focus node v of (s, φ, τ) as the triples (v, p, o) in $\llbracket \Sigma \rrbracket$ whose predicate is among those that validation of φ would inspect, i.e., $p \in P(\varphi)$ (Section 2.2), together with triples witnessing τ . For each $(s, \varphi, \tau) \in \mathcal{H}$: Σ'_{RE} generates exactly the *relevant triples* of each focus node. Σ'_{CO} generates exactly the *relevant triples* of each focus node v with $\llbracket \Sigma \rrbracket, v \models \varphi$; no triples are generated for focus nodes with $\llbracket \Sigma \rrbracket, v \not\models \varphi$. Σ'_{EV} generates exactly the *relevant triples* of each focus node v with $\llbracket \Sigma \rrbracket, v \models \varphi$ that constitute the minimal deterministic evidence for its conformance (cf. $B(v, \llbracket \Sigma \rrbracket, \varphi \wedge \tau)$, Section 2.2). By applying $\neg\varphi$, Σ'_{CO} and Σ'_{EV} symmetrically generate triples of non-conforming focus nodes, supporting why-not-provenance.

Table 1. Coverage and classification of SHACL constructs. Co.: ✓ supported, ✗ partially, ✗ not supported. Func.: RML function used (– if none).

Construct	Co.	Func.	Construct	Co.	Func.	Construct	Co.	Func.
sh:targetClass	✓	–	sh:targetSubj.	✗	–	sh:targetObj.	✗	–
sh:targetNode	✗	–	sh:class	✗	–	sh:datatype	✓	–
sh:nodeKind	✓	–	sh:languageIn	✓	–	sh:minInclusive	✓	$f_{\min\text{Incl}}$
sh:maxInclusive	✓	$f_{\max\text{Incl}}$	sh:minExclusive	✓	$f_{\min\text{Excl}}$	sh:maxExclusive	✓	$f_{\max\text{Excl}}$
sh:minLength	✓	$f_{\min\text{Len}}$	sh:maxLength	✓	$f_{\max\text{Len}}$	sh:pattern	✓	f_{pattern}
sh:in	✓	f_{in}	sh:uniqueLang	✓	–	sh:minCount*	✓	$f_{\text{count}_{\geq}}$
sh:maxCount*	✓	$f_{\text{count}_{\leq}}$	sh:hasValue*	✓	$f_{\text{count}_{\geq, \text{eq}}}$	sh:equals*	✓	f_{eq}
sh:disjoint*	✓	f_{disj}	sh:lessThan*	✓	f_{lt}	sh:lessThanOrEq.*	✓	f_{leq}
sh:and	✗	f_{and}	sh:or	✗	f_{or}	sh:not	✗	f_{not}
sh:xone	✗	$f_{\text{and, or, not}}$	sh:closed	✓	–	sh:ignoredProp.	✓	–
sh:node	✗	–	sh:property	✓	–	sh:qualValShape*	✗	$f_{\text{count}_{\geq, \leq}}$
sh:qualMinCount*	✗	$f_{\text{count}_{\geq}}$	sh:qualMaxCount*	✗	$f_{\text{count}_{\leq}}$	sh:path	✗	–

* Affected by injectivity assumption. Full function semantics in Appendix C.

3.1 Compilability of SHACL into RML and Constraint Coverage

We explain the necessity of the assumptions for supporting certain constraints under RML engine’s record-at-a-time processing model. Table 1 annotates the affected constraints. SHACL validation evaluates each focus node v against all of v ’s relevant triples, but in RML multiple records may contribute triples to the same focus node. Without injectivity (Remark 4), two table rows may both produce `:article1` node, while a constraint `sh:maxCount 1` on `:year` then cannot be evaluated during mapping execution, as the engine has no cross-record access to aggregate the `:year` triples. Without class coherence (Remark 5), a tm generating `:year` for `:article1` without declaring its class, while another tm' generating the same node declares, cannot be identified at the mapping stage. Even under these assumptions, constraints cannot be directly compiled into the aligned tm , as multiple tms may contribute triples to the same focus node. We address this by merging aligned tms into a single tm in Section 3.2.

The partially supported constructs in Table 1 are fully supported when φ' within $\forall p.\varphi'$ or $\#_n^m p.\varphi'$ is *path free*, i.e., a Boolean combination of `test` and `hasValue` after unfolding any referenced shapes, and restricted otherwise. Validating these requires access to the node o' reached via p' from the object node o generated by a om , which cannot be supported due to two obstacles: (a) if om is not a referencing object map, the tm' generating o' with p' cannot be identified; (b) even if om is a referencing object map and the parent tm' can be located, the conformance result of o' in tm' cannot be propagated back to tm during execution. Runtime functions cannot be injected on a referencing object map [16], restricting the mappings we can compile. The same obstacles restrict property paths to direct predicate paths (Remark 2). Under `sh:targetSubjectsOf` p and `sh:targetObjectsOf` p , φ is restricted to constraints on the focus node itself, as the triples maps generating its remaining triples cannot be identified: those without a pom with predicate p may generate the same subject as those with one, and those generating the object of p as a subject cannot be located either.

3.2 Mapping Consolidation

Mapping consolidation aligns each shape definition with triples maps in Σ , prunes irrelevant predicate-object maps, and for Σ'_{CO} and Σ'_{EV} merges the aligned triples maps (if multiple) into a single triples map per shape definition.

Since a triples map may match the target declarations of multiple shape definitions, we instantiate a dedicated copy per match so that each shape definition is compiled independently on its own triples map.

Definition 10 (Triples Map Alignment). Given a shape definition (s, φ, τ) and a mapping Σ , the *alignment set* $\mathcal{A} = \text{align}((s, \varphi, \tau), \Sigma)$ ($\emptyset$ for $\tau = \neg\top$) is:

$$\begin{cases} \{\text{inst}(tm, c) \mid tm \in \Sigma, \exists (\text{const}(a), \text{const}(c)) \in \text{poms}(tm)\} & \tau = \geq_1 \text{rdf:type.hasValue}(c) \\ \{\text{inst}(tm, p) \mid tm \in \Sigma, \exists (\text{const}(p), \cdot) \in \text{poms}(tm)\} & \tau = \geq_1 p.\top \\ \{\text{inst}(tm, p) \mid tm \in \Sigma, \exists (\text{const}(p), \cdot) \in \text{poms}(tm)\} & \tau = \geq_1 p^-. \top \end{cases}$$

where $\text{inst}(tm, \cdot)$ creates a dedicated copy of tm with a unique identifier.

Next, predicate-object maps whose predicates are not in $P(\varphi)$ are pruned from each aligned triples map, while $\text{preds_orig}(tm)$ records the set of all predicate IRIs in $\text{poms}(tm)$ before pruning for subsequent closed validation.

Definition 11 (Predicate-Object Map Pruning). For each $tm \in \mathcal{A}$, we define $\text{prune}(tm, (s, \varphi, \tau)) = (id, ls, sm, poms')$, where $poms'$ is:

$$\begin{cases} \{(pm, om) \in \text{poms}(tm) \mid pm = \text{const}(a) \wedge om = \text{const}(c)\} \\ \cup \{(pm, om) \in \text{poms}(tm) \mid pm = \text{const}(p'), p' \in P(\varphi)\} & \tau = \geq_1 \text{rdf:type.hasValue}(c) \\ \{(pm, om) \in \text{poms}(tm) \mid pm = \text{const}(p)\} \\ \cup \{(pm, om) \in \text{poms}(tm) \mid pm = \text{const}(p'), p' \in P(\varphi)\} & \tau = \geq_1 p.\top \\ \{(pm, om) \in \text{poms}(tm) \mid pm = \text{const}(p)\} & \tau = \geq_1 p^-. \top \end{cases}$$

At this point, for each $(s, \varphi, \tau) \in \mathcal{H}$, alignment and pruning yield $\mathcal{A}'(s, \varphi, \tau) = \{\text{prune}(tm, (s, \varphi, \tau)) \mid tm \in \text{align}((s, \varphi, \tau), \Sigma)\}$. For Σ'_{CO} and Σ'_{EV} , when multiple triples maps remain in $\mathcal{A}'$, they are merged into a single triples map.

This merge requires combining their logical sources into a single source, which we achieve here with RML views [2,23], constructing $\text{rmlView}(q)$ that unifies all sources into a virtual table, renaming references from different triples maps to avoid conflicts and joining records on subject IRI. We define $\text{sqljoin}(\mathcal{A}')$ to construct this SQL query: for each $tm_i \in \mathcal{A}'$, a subquery computes the subject IRI as a `__sid__` column from its logical source and renames all references rf in $\text{poms}(tm_i)$ to $\sigma_i(rf) = \text{t}_i_rf$. All subqueries are joined via `FULL OUTER JOIN` on `__sid__`, aligning records generating the same subject IRI to the same row. A concrete example of sqljoin , sid , and σ_i is given in Appendix B.

We then define $\text{merge}(\mathcal{A}') = (id', ls', sm', poms')$ for $|\mathcal{A}'| > 1$, where $ls' = \text{rmlView}(\text{sqljoin}(\mathcal{A}'))$, $sm' = \text{ref}("__sid__", \text{IRI})$ preserving injectivity, and $poms' = \bigcup_{i=1}^n \text{rename}(\text{poms}(tm_i), \sigma_i)$, which renames each reference rf with $\sigma_i(rf)$.

We combine the above steps into a single function $\text{consolidate}(\Sigma, \mathcal{H}, \text{mode})$, as described in Algorithm 1. For $\text{mode} = \text{RE}$, it returns $\Sigma'_{\text{RE}} = \bigcup_{(s, \varphi, \tau) \in \mathcal{H}} \mathcal{A}'(s, \varphi, \tau)$,

Algorithm 1: consolidate($\Sigma, \mathcal{H}, mode$)	Algorithm 2: rewrite($\Sigma, \mathcal{H}, mode$)
Input: $\Sigma, \mathcal{H}, mode \in \{RE, CO, EV\}$ Output: Σ'_{RE} or Π 1 $\Sigma'_{RE} \leftarrow \emptyset; \Pi \leftarrow \emptyset;$ 2 foreach $(s, \varphi, \tau) \in \mathcal{H}$ do 3 $\mathcal{A} \leftarrow \text{align}((s, \varphi, \tau), \Sigma);$ <i>// triples maps matching τ</i> 4 $\mathcal{A}' \leftarrow \{\text{prune}(tm, (s, \varphi, \tau)) \mid tm \in \mathcal{A}\};$ 5 if $\mathcal{A}' = \emptyset$ then continue ; 6 if $mode = RE$ then $\Sigma'_{RE} \leftarrow \Sigma'_{RE} \cup \mathcal{A}';$ 7 else if $ \mathcal{A}' = 1$ then $\Pi(s, \varphi, \tau) \leftarrow tm \in \mathcal{A}';$ 8 else $\Pi(s, \varphi, \tau) \leftarrow \text{merge}(\mathcal{A}');$ <i>// one triples map per shape definition</i> 9 if $mode = RE$ then return $\Sigma'_{RE};$ 10 $\text{renameparent}(\Pi);$ <i>// update referencing object maps</i> 11 return $\Pi;$	Input: $\Sigma, \mathcal{H}, mode \in \{RE, CO, EV\}$ Output: Σ' 1 $R \leftarrow \text{consolidate}(\Sigma, \mathcal{H}, mode); \Sigma' \leftarrow \emptyset;$ 2 if $mode = RE$ then return $R;$ 3 foreach $(s, \varphi, \tau) \in \text{dom}(R)$ do 4 $tm \leftarrow R(s, \varphi, \tau); poms' \leftarrow \emptyset;$ 5 foreach $map \in \text{fmaps}(tm, \tau)$ do 6 $\varepsilon_v \leftarrow \text{compile}_v(\varphi, map, tm, \mathcal{H});$ 7 if $\varepsilon_v = \perp$ then $tm \leftarrow \text{drop}(tm, map);$ 8 else if $\varepsilon_v \neq \top$ then $map \leftarrow \text{passif}(map, \varepsilon_v);$ 9 if $\varepsilon_v \neq \perp \wedge mode = EV$ then $poms' \leftarrow$ $poms' \cup \text{compile}_n(\varphi \wedge \tau, map, tm, \mathcal{H});$ 10 if $tm = \perp$ then continue ; 11 if $mode = EV$ then $poms(tm) \leftarrow \text{dedup}(poms');$ 12 $\Sigma' \leftarrow \Sigma' \cup \{tm\};$ 13 return $\Sigma';$

the union of all pruned triples maps. For $mode \in \{CO, EV\}$, it further merges each $\mathcal{A}'$ with $|\mathcal{A}'| > 1$ into a single triples map tm' with $\text{preds_orig}(tm') = \bigcup_{i=1}^n \text{preds_orig}(tm_i)$, producing a partial function $\Pi : (s, \varphi, \tau) \mapsto tm$ for subsequent compilation, and applies renameparent to update referencing object maps.

Example 4. Continuing Example 3. Alignment produces $\mathcal{A}(:\text{ArticleShape}, \varphi_1, \tau_1) = \{tm'_{art}, tm'_{jour}\}$ as both declare $:\text{Article}$, and $\mathcal{A}(:\text{JournalShape}, \varphi_2, \tau_2) = \{tm''_{jour}\}$. Pruning then discards $:\text{title}$ and $:\text{journal}$ for $:\text{ArticleShape}$ with $P(\varphi_1) = \{:\text{year}\}$, and $:\text{title}$ for $:\text{JournalShape}$ with $P(\varphi_2) = \{:\text{year}, :\text{journal}\}$. Σ'_{RE} contains all three pruned triples maps. For Σ'_{CO} and Σ'_{EV} , $\mathcal{A}'(:\text{ArticleShape}, \varphi_1, \tau_1)$ is merged into a single triples map over $t1_year$ and $t2_year$, while the $\Pi(:\text{JournalShape}, \varphi_2, \tau_2)$ remains unchanged. Full rmView in Appendix B.

3.3 Compiling SHACL Constraints into RML

Given the consolidated mappings, Table 2 defines a validation compilation that combines static analysis with runtime function injection on the term map generating each focus node so that only the *relevant triples* of conforming focus nodes are generated (Σ'_{CO}), and Table 3 defines a neighborhood compilation that further filters triples to the minimal deterministic evidence (Σ'_{EV}).

To this end, we encode each constraint as a function-valued term map $\text{fexec}(fe)$ (Definition 7), abbreviated by f throughout. Table 1 lists the function for each construct, whose behavior follows its name (e.g., f_{eq} checks equality, $f_{count_{\geq}}$ checks whether the number of values is at least n). Constraints resolved statically, including sh:datatype , sh:nodeKind , sh:closed , and language-relevant, do not require functions. We define below the output domain and input parameters of all referenced f . Full semantics of each f in Appendix C.

Definition 12 (Constraint Functions). A *constraint function* f evaluates on a record r to a value in $\{\text{true}, \text{false}, \text{none}\}$, where true and false are xsd:boolean

Table 2. Validation compilation $\text{compile}_v(\varphi, \text{map}, \text{tm}, \mathcal{H})$, which recursively translates φ into a rewriting decision operated via $\hat{\wedge}$, $\hat{\vee}$, $\hat{\neg}$ (Definition 13).

φ	$\text{compile}_v(\varphi, \text{map}, \text{tm}, \mathcal{H}) \rightarrow \{\top, \perp, f\}$
$\top$	$\top$
$\perp$	$\perp$
$\varphi_1 \wedge \varphi_2$	$\text{compile}_v(\varphi_1, \text{map}, \text{tm}, \mathcal{H}) \hat{\wedge} \text{compile}_v(\varphi_2, \text{map}, \text{tm}, \mathcal{H})$
$\varphi_1 \vee \varphi_2$	$\text{compile}_v(\varphi_1, \text{map}, \text{tm}, \mathcal{H}) \hat{\vee} \text{compile}_v(\varphi_2, \text{map}, \text{tm}, \mathcal{H})$
$\neg \varphi'$	$\hat{\neg} \text{compile}_v(\varphi', \text{map}, \text{tm}, \mathcal{H})$
$\text{hasShape}(s)$	$\text{compile}_v(\text{exp}(s, \mathcal{H}), \text{map}, \text{tm}, \mathcal{H})$
$\text{hasValue}(c)$	$f_{\text{eq}}(\text{map}, \text{const}(c))$
$\text{test}(\text{nodeKind}(k))$	$\top$ if declared or inferred term type of map is compatible with k ; $\perp$ o.w.
$\text{test}(\text{datatype}(d))$	$\top$ if map declares or defaults to datatype d ; $\perp$ o.w.
$\text{test}(\text{langIn}(L))$	$\top$ if map declares language tag $\ell \in L$; $\perp$ o.w.
$\text{test}(t(v)), t \in \Omega_V$	$f_t(\text{map}, \text{const}(v))$ (cf. Table 1)
$\forall p. \varphi'$	$\hat{\wedge}_{(\cdot, \text{om}) \in \text{poms}_p} \text{compile}_v(\varphi', \text{om}, \text{tm}, \mathcal{H})$ if $\text{poms}_p \neq \emptyset$; $\top$ if $\text{poms}_p = \emptyset$
$\geq_n p. \varphi'$	$f_{\text{count} \geq}(\{(\text{om}, \text{compile}_v(\varphi', \text{om}, \text{tm}, \mathcal{H})) \mid (\cdot, \text{om}) \in \text{poms}_p\}, n)$ if $\text{poms}_p \neq \emptyset$; $\top$ if $n=0$; $\perp$ o.w.
$\leq_n p. \varphi'$	$f_{\text{count} \leq}(\{(\text{om}, \text{compile}_v(\varphi', \text{om}, \text{tm}, \mathcal{H})) \mid (\cdot, \text{om}) \in \text{poms}_p\}, n)$ if $\text{poms}_p \neq \emptyset$; $\top$ if $\text{poms}_p = \emptyset$
$\text{eq}(p, p')$	$f_{\text{eq}}(\text{om}_p, \text{om}_{p'})$
$\text{disj}(p, p')$	$f_{\text{disj}}(\text{om}_p, \text{om}_{p'})$
$\text{lessThan}(p, p')$	$f_{\text{lt}}(\text{om}_p, \text{om}_{p'})$
$\text{lessThanEq}(p, p')$	$f_{\text{leq}}(\text{om}_p, \text{om}_{p'})$
$\text{uniqueLang}(p)$	$\top$ if no two om from $(\cdot, \text{om}) \in \text{poms}_p$ declare the same language tag; $\perp$ o.w.
$\text{closed}(P)$	$\top$ if $\text{preds_orig}(\text{tm}) \subseteq P$; $\perp$ o.w.

$\text{exp}(s, \mathcal{H})$ denotes the φ of s in $\mathcal{H}$, defaulting to $\top$. f with all-constant arguments is statically evaluated. om_p denotes the object maps $(\text{om}_1, \dots, \text{om}_k)$ of $\text{poms}_p(\text{tm})$.

literals and **none** denotes an absent value. We write all arguments of f as $f(\text{tmap}_1, \dots, \text{tmap}_k)$, abbreviating the parameter IRIs in $(\text{par}, \text{tmap})$.

Additionally, the compilation requires a mechanism to conditionally suppress triple generation based on the boolean conformance result from f : $f_{\text{passif}}(\text{tmap}, f)$ returns $\text{eval}(\text{tmap}, r)$ unless f evaluates to **false**, in which case it returns **none**, enabling the subject or object map to output **none** for non-conforming nodes.

Some constraints are resolved statically at compile time, while others require runtime validation via functions. Static resolution relies on what a term map declares, and a term map may generate no triple on a record. We do not consider the degenerate case where a constraint holds vacuously because the relevant triples are absent or coincide. A rewriting decision provides reduction operators to combine multiple decisions into a single actionable operation.

Definition 13 (Rewriting Decision Reduction). The compilation of each constraint yields a *rewriting decision* ε : $\perp$ (statically violates), $\top$ (statically satisfies), or a runtime check f (a constraint function). Decisions are resolved via commutative reduction operators: $\top \hat{\wedge} \top = \top$, $\perp \hat{\wedge} \perp = \perp$, $\perp \hat{\wedge} \top = \perp$, $\top \hat{\wedge} f = f$, $\perp \hat{\wedge} f = \perp$, $f_1 \hat{\wedge} f_2 = f_{\text{and}}(f_1, f_2)$; $\top \hat{\vee} \top = \top$, $\perp \hat{\vee} \perp = \perp$, $\perp \hat{\vee} \top = \top$, $\top \hat{\vee} f = \top$, $\perp \hat{\vee} f = f$, $f_1 \hat{\vee} f_2 = f_{\text{or}}(f_1, f_2)$; $\hat{\neg} \top = \perp$, $\hat{\neg} \perp = \top$, $\hat{\neg} f = f_{\text{not}}(f)$.

To produce Σ'_{CO} , each constraint must be translated into a rewriting decision that contributes to determining whether a focus node's triples are generated or suppressed, based on whether values from a term map conform to a φ .

Definition 14 (Validation Compilation). Given a shape definition (s, φ, τ) , a $tm = \Pi(s, \varphi, \tau)$, and a term map map generating a focus node, either $\text{sm}(tm)$ or an object map of $\text{poms}_p(tm)$, the *validation compilation* $\text{compile}_v(\varphi, map, tm, \mathcal{H})$, defined inductively on φ in Table 2, yields a rewriting decision ε_v , aligning each case with the conformance semantics of [10] (reproduced in Appendix A.1).

For $mode \in \{\text{CO}, \text{EV}\}$, Algorithm 2 then applies compile_v to each $tm = \Pi(s, \varphi, \tau)$. Functions are injected into $\text{fmaps}(tm, \tau)$, the term maps generating each focus node of τ : the object maps of $\text{poms}_p(tm)$ when $\tau = \geq_1 p^-. \top$, and $\text{sm}(tm)$ otherwise. $\text{drop}(tm, map)$ removes map from tm : the subject map, or the predicate-object map whose object map is map . It returns $\perp$ if no subject map or no predicate-object map remains, and tm otherwise. For each $map \in \text{fmaps}(tm, \tau)$: if $\varepsilon_v = \perp$, $\text{drop}(tm, map)$ is applied; if ε_v is some f , map is replaced by $f_{\text{passif}}(map, f)$; if $\varepsilon_v = \top$, map is unchanged. For $mode = \text{CO}$, it returns the resulting triples maps as Σ'_{CO} . For $mode = \text{EV}$, each pom is further filtered to retain only the minimal deterministic evidence. A $\text{resolve}_n(pom, \varepsilon)$ function determines, based on the validation decision ε from corresponding compile_v , whether the pom is retained unfiltered ($\varepsilon = \top$), retained with a runtime filter via $f_{\text{passif}}(om, f)$ on its object map ($\varepsilon = f$), or removed entirely ($\varepsilon = \perp$). $\text{inject}(\text{poms}_p, f)$ likewise replaces each object map by $f_{\text{passif}}(om, f(om))$.

Definition 15 (Neighborhood Compilation). Given a shape definition (s, φ, τ) , a $tm = \Pi(s, \varphi, \tau)$, and a term map map generating a focus node, the *neighborhood compilation* $\text{compile}_n(\varphi \wedge \tau, map, tm, \mathcal{H})$, defined inductively on φ in Table 3, yields the set of pom constituting the minimal deterministic evidence. At each inductive step, compile_v determines the validation decision for each pom , which resolve_n uses to filter accordingly. Each case aligns with the *neighborhood* definition of [10] (reproduced in Appendix A.2).

Following Algorithm 2, compile_n produces the predicate-object maps poms' , accumulated over all such term maps. Since different inductive cases may yield the same pom both with and without a function on its object map, dedup retains the version without, as it subsumes the other. These triples maps constitute Σ'_{EV} .

Example 5. Continuing Example 4. For φ_1 , compile_v enters the $\forall:\text{year}$ rule, yielding $\varepsilon_v = f_{\text{and}}(f_{\text{maxIncl}}(\text{ref}(\text{t1_year}), \text{const}(2026)), f_{\text{maxIncl}}(\text{ref}(\text{t2_year}), \text{const}(2026)))$; the subject map is replaced by $f_{\text{passif}}(sm, \varepsilon_v)$. For φ_2 , compile_v resolves statically to $\top$ (term types match). compile_n retains all poms unfiltered for both shapes, so $\Sigma'_{\text{CO}} = \Sigma'_{\text{EV}}$. At runtime, :p1 is suppressed while :p2 generates triples.

3.4 Correctness

We establish correctness with respect to the provenance semantics of [10]. Let $G = \llbracket \Sigma \rrbracket$ and $\Pi = \text{consolidate}(\Sigma, \mathcal{H}, mode)$. We write $\llbracket \Sigma'_{mode}(s, \varphi, \tau) \rrbracket$ for the triples generated for (s, φ, τ) in Σ'_{mode} . Full proofs in Appendix D.

Table 3. Neighborhood compilation $\text{compile}_n(\varphi, \text{map}, \text{tm}, \mathcal{H})$.

φ	$\text{compile}_n(\varphi, \text{map}, \text{tm}, \mathcal{H}) \rightarrow \text{poms}'$
$\varphi_1 \wedge \varphi_2$	$\text{compile}_n(\varphi_1, \text{map}, \text{tm}, \mathcal{H}) \cup \text{compile}_n(\varphi_2, \text{map}, \text{tm}, \mathcal{H})$
$\varphi_1 \vee \varphi_2$	$\bigcup_{(\cdot, \text{om}) \in \text{compile}_n(\varphi_1, \text{map}, \text{tm}, \mathcal{H})} \text{resolve}_n((\cdot, \text{om}), \varepsilon_1) \cup$ $\bigcup_{(\cdot, \text{om}) \in \text{compile}_n(\varphi_2, \text{map}, \text{tm}, \mathcal{H})} \text{resolve}_n((\cdot, \text{om}), \varepsilon_2);$ $\varepsilon_i = \text{compile}_v(\varphi_i, \text{map}, \text{tm}, \mathcal{H}), i \in \{1, 2\}$
$\text{hasShape}(s)$	$\text{compile}_n(\text{exp}(s, \mathcal{H}), \text{map}, \text{tm}, \mathcal{H})$
$\neg \text{hasShape}(s)$	$\text{compile}_n(\neg \text{exp}(s, \mathcal{H}), \text{map}, \text{tm}, \mathcal{H})$
$\forall p. \varphi'$	poms_p
$\geq_n p. \varphi'$	$\bigcup_{(\cdot, \text{om}) \in \text{poms}_p} \text{resolve}_n((\cdot, \text{om}), \varepsilon_{\text{om}}); \varepsilon_{\text{om}} = \text{compile}_v(\varphi', \text{om}, \text{tm}, \mathcal{H})$
$\leq_n p. \varphi'$	$\bigcup_{(\cdot, \text{om}) \in \text{poms}_p} \text{resolve}_n((\cdot, \text{om}), \hat{\varepsilon}_{\text{om}}); \varepsilon_{\text{om}} = \text{compile}_v(\varphi', \text{om}, \text{tm}, \mathcal{H})$
$\geq_n p^-. \top$	$\bigcup_{(\cdot, \text{om}) \in \text{poms}_p, \text{om}=\text{map}} \text{resolve}_n((\cdot, \text{om}), \varepsilon_{\text{om}}); \varepsilon_{\text{om}} = \text{compile}_v(\top, \text{om}, \text{tm}, \mathcal{H})$
$\text{eq}(p, p')$	$\text{poms}_p \cup \text{poms}_{p'}$
$\neg \text{eq}(p, p')$	$\text{inject}(\text{poms}_p, f_{\text{disj}}(\text{om}, \text{om}_{p'})) \cup \text{inject}(\text{poms}_{p'}, f_{\text{disj}}(\text{om}, \text{om}_p))$
$\neg \text{disj}(p, p')$	$\text{inject}(\text{poms}_p, f_{\text{eq}}^{\exists}(\text{om}, \text{om}_{p'})) \cup \text{inject}(\text{poms}_{p'}, f_{\text{eq}}^{\exists}(\text{om}, \text{om}_p))$
$\neg \text{lessThan}(p, p')$	$\text{inject}(\text{poms}_p, f_{\text{eq}}^{\exists}(\text{om}, \text{om}_{p'})) \cup \text{inject}(\text{poms}_{p'}, f_{\text{eq}}^{\exists}(\text{om}, \text{om}_p))$
$\neg \text{lessThanEq}(p, p')$	$\text{inject}(\text{poms}_p, f_{\text{lt}}^{\exists}(\text{om}, \text{om}_{p'})) \cup \text{inject}(\text{poms}_{p'}, f_{\text{lt}}^{\exists}(\text{om}, \text{om}_p))$
$\neg \text{uniqueLang}(p)$	$\{(pm, \text{om}) \in \text{poms}_p \mid \exists (pm', \text{om}') \in \text{poms}_p \setminus \{(pm, \text{om})\} : \text{lang}(\text{om}) = \text{lang}(\text{om}')\}$
$\neg \text{closed}(P)$	$\{(pm, \text{om}) \in \text{poms}(\text{tm}) \mid pm = \text{const}(p), p \notin P\}$

Constructs not listed return $\emptyset$. The $\geq_n p^-. \top$ entry arises for target declarations.

Lemma 1 (Consolidation Completeness). *For each $(s, \varphi, \tau) \in \mathcal{H}$ and every v with $G, v \models \tau$: every relevant triple of v is in $\llbracket \Sigma'_{\text{RE}}(s, \varphi, \tau) \rrbracket$ and in $\llbracket \Pi(s, \varphi, \tau) \rrbracket$.*

Proof (Proof sketch). By Definition 10, every tm in Σ with a pom matching τ is included in the alignment, covering all tms generating *relevant triples* for any focus node v . By Definition 11, pruning retains the target-witnessing poms and those with predicates in $P(\varphi)$, covering all predicates of *relevant triples*. Merging via FULL OUTER JOIN preserves all resources from the aligned tms .

Lemma 2 (Consolidation Soundness). *For each $(s, \varphi, \tau) \in \mathcal{H}$: every triple in $\llbracket \Sigma'_{\text{RE}}(s, \varphi, \tau) \rrbracket$ and in $\llbracket \Pi(s, \varphi, \tau) \rrbracket$ is a relevant triple of a v with $G, v \models \tau$.*

Proof (Proof sketch). By the same alignment and pruning as Lemma 1. Only tms matching τ are aligned, and only poms with p in $P(\varphi)$ retained, ensuring every generated triple is a *relevant triple*. Merging introduces no new triples.

Lemma 3 (Validation Compilation Completeness). *For each $(s, \varphi, \tau) \in \mathcal{H}$ and every v with $G, v \models \tau$ and $G, v \models \varphi$: every relevant triple of v is in $\llbracket \Sigma'_{\text{CO}}(s, \varphi, \tau) \rrbracket$.*

Proof (Proof sketch). By Lemma 1, every *relevant triple* of v is in $\llbracket \Pi(s, \varphi, \tau) \rrbracket$. The validation compilation (Table 2) translates φ into a rewriting decision ε_v by structural induction on φ , aligning each case with the conformance semantics of Appendix A.1 [10]. Since v conforms to φ , $\varepsilon_v \neq \perp$ for the term map generating v , so none of its *relevant triples* is dropped. If ε_v is some f , f_{passif} evaluates to true for v , preserving all its triples.

Lemma 4 (Validation Compilation Soundness). *As above, but for $G, v \not\models \varphi$: no relevant triple of v is in $\llbracket \Sigma'_{\text{CO}}(s, \varphi, \tau) \rrbracket$.*

Proof (Proof sketch). By the same structural induction as Lemma 3. Since v does not conform to φ , either $\varepsilon_v = \perp$ and the term map generating v is dropped, or ε_v is some f and f_{passif} evaluates to false, so no *relevant triple* of v is generated.

Lemma 5 (Neighborhood Compilation Completeness). *For each $(s, \varphi, \tau) \in \mathcal{H}$ and every v with $G, v \models \tau$ and $G, v \models \varphi$: every triple in $B(v, G, \varphi \wedge \tau)$ is in $\llbracket \Sigma'_{\text{EV}}(s, \varphi, \tau) \rrbracket$.*

Proof (Proof sketch). By Lemma 3, every relevant triple of conforming v is in $\llbracket \Sigma'_{\text{CO}} \rrbracket$. The neighborhood compilation (Table 3) determines which poms contribute to $B(v, G, \varphi \wedge \tau)$ by structural induction on φ , aligning each case with the neighborhood definition of Appendix A.2 [10]. Every triple in $B(v, G, \varphi \wedge \tau)$ is retained by the corresponding pom or injected filter.

Lemma 6 (Neighborhood Compilation Soundness). *As above: every triple of v in $\llbracket \Sigma'_{\text{EV}}(s, \varphi, \tau) \rrbracket$ is in $B(v, G, \varphi \wedge \tau)$.*

Proof (Proof sketch). By the same structural induction as Lemma 5. Poms not contributing to $B(v, G, \varphi \wedge \tau)$ are removed, and those requiring value-level filtering are injected with functions returning **none** for non-contributing triples, ensuring no triple outside $B(v, G, \varphi \wedge \tau)$ is generated.

Theorem 3 (Correctness). *$\llbracket \Sigma'_{\text{RE}} \rrbracket$ consists of exactly the relevant triples of each focus node; for each $(s, \varphi, \tau) \in \mathcal{H}$, $\Pi(s, \varphi, \tau)$ in Σ'_{CO} generates exactly the relevant triples of each conforming focus node and none of those of non-conforming ones; $\llbracket \Sigma'_{\text{EV}} \rrbracket = \text{Frag}(G, \mathcal{H})$.*

Proof. The first claim follows from Lemmas 1 and 2. The second follows from Lemmas 1–4. The third follows from Lemmas 1–6 and Definition 5.

Corollary 1 (Sufficiency). *For each $(s, \varphi, \tau) \in \mathcal{H}$ and every v with $G, v \models \tau$ and $G, v \models \varphi$: v conforms to φ in $\llbracket \Sigma'_{\text{RE}}(s, \varphi, \tau) \rrbracket$, $\llbracket \Sigma'_{\text{CO}}(s, \varphi, \tau) \rrbracket$, and $\llbracket \Sigma'_{\text{EV}}(s, \varphi, \tau) \rrbracket$.*

Proof. By Theorem 3, the three facets select progressively fewer of the same *relevant triples*, so $B(v, G, \varphi \wedge \tau) \subseteq \llbracket \Sigma'_{\text{EV}}(s, \varphi, \tau) \rrbracket \subseteq \llbracket \Sigma'_{\text{CO}}(s, \varphi, \tau) \rrbracket \subseteq \llbracket \Sigma'_{\text{RE}}(s, \varphi, \tau) \rrbracket \subseteq G$, and the claim follows by Theorem 1.

Corollary 2 (Conformance). *If $G \models \mathcal{H}$, then $\llbracket \Sigma'_{\text{RE}} \rrbracket$, $\llbracket \Sigma'_{\text{CO}} \rrbracket$, $\llbracket \Sigma'_{\text{EV}} \rrbracket$ all $\models \mathcal{H}$.*

Proof. Σ'_{EV} : $\llbracket \Sigma'_{\text{EV}} \rrbracket = \text{Frag}(G, \mathcal{H})$ by Theorem 3, and the result follows from Theorem 2. Σ'_{CO} and Σ'_{RE} : by Theorem 3, $\text{Frag}(G, \mathcal{H}) \subseteq \llbracket \Sigma'_{\text{CO}} \rrbracket \subseteq \llbracket \Sigma'_{\text{RE}} \rrbracket \subseteq G$, and with $G \models \mathcal{H}$ and monotone targets, every focus node conforms by Theorem 1.

4 Implementation and Experimental Validation

We evaluate our implementation to answer: **RQ1** (Correctness): Does the implementation correctly produce the expected output for each of the three facets?

RQ2 (Efficiency): How does the performance of our approach compare to the baseline of full materialization followed by post-hoc shape fragment extraction?

We implement an open-source system, SHACL2RML¹. It takes a SHACL schema and an RML mapping as input and produces a rewritten mapping for a specified facet mode, following Algorithms 1 and 2. A `-not` flag negates each shape definition’s φ to $\neg\varphi$ (expanded to NNF), enabling users to directly generate non-conforming triples. During rewriting, the input schema is automatically partitioned into a supported schema covering the supported fragment and a deferred schema for the remainder, both are returned to the user, ensuring compatibility with any real-world SHACL shapes.

The time complexity of the algorithm for Σ'_{RE} is in $O(|\mathcal{H}| \cdot |\Sigma| \cdot \max |\text{poms}|)$, where $|\mathcal{H}|$ is the number of shape definitions, $|\Sigma|$ the number of triples maps, and $\max |\text{poms}|$ the largest number of predicate-object maps in any triples map. Σ'_{CO} and Σ'_{EV} each add $O(|\mathcal{H}| \cdot |\varphi| \cdot \max |\text{poms}|)$, where $|\varphi|$ is the number of constraints in φ after unfolding shape references. All stages in polynomial time.

All experiments in this section ran on a machine with dual Intel Xeon Platinum 8360Y CPUs at 2.40 GHz, each job allocated 16 cores and 128 GiB RAM, except the baseline at the two largest scales which are allocated 768 GiB.

Correctness (RQ1). We construct 50 test cases covering all supported SHACL constructs and their combinations (see Appendix E for statistics), each with source data, RML mappings, and expected output for all three facets under both $\mathcal{H}$ and $\neg\mathcal{H}$. The EV mode test cases were independently verified against pySHACL-fragments [9], the implementation from [10] that extracts shape fragments from a materialized $\llbracket \Sigma \rrbracket$, though it does not implement all SHACL constraints. Our pipeline first applies SHACL2RML to compile $\mathcal{H}$ or $\neg\mathcal{H}$ and Σ into a rewritten Σ' , then executes it via Morph-KGC [3], one of the fastest RML engines [27] and the only one supporting the RML views [23] we adopted. Each test case is executed on CSV, JSON, XML, and relational sources, yielding 400 executions. SHACL2RML completes every compilation in 0.04 seconds. All test cases pass except two XML cases that require RML views, which Morph-KGC does not yet support for XML. SDM-RDFizer [15] supports the related RML logical views [21], to which our merge of logical sources can be transferred.

Efficiency (RQ2). We evaluate our pipeline’s EV mode under $\neg\mathcal{H}$, which extracts the minimal evidence for non-conformance of each focus node during construction, against the baseline of Morph-KGC followed by pySHACL-fragments [9], which extracts $\text{Frag}(\llbracket \Sigma \rrbracket, \neg\mathcal{H})$ from a materialized graph. We evaluate on the GTFS benchmark [6] with CSV sources (correctness across other formats is confirmed by RQ1), RML mappings, and a SHACL schema manually created based on the GTFS ontology and mappings, at five scales. Figure 1 reports the results. Our pipeline is consistently up to three orders of magnitude faster with proportionally lower memory than the baseline, whose performance is attributable to its pySHACL-based implementation, which loads the entire graph into memory for validation. Although pySHACL-fragments does not support $\neg\mathcal{H}$ for this schema, the output under $\mathcal{H}$ is identical between our pipeline and the baseline at all scales, and the output counts confirm $\llbracket \Sigma'(\mathcal{H}) \rrbracket \cup \llbracket \Sigma'(\neg\mathcal{H}) \rrbracket = \llbracket \Sigma \rrbracket$.

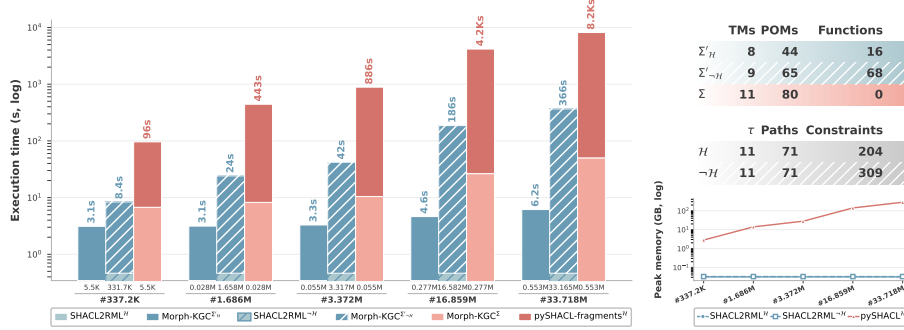


Fig. 1. Execution time (left) and peak memory (right) of our pipeline (blue) vs. the baseline (red) on GTFS at five scales. Three bars per scale: ours under $\mathcal{H}$, ours under $\neg\mathcal{H}$, and baseline under $\mathcal{H}$. $\neg\mathcal{H}$ is not supported by pySHACL-fragments for this schema, hence no bar. Output triple counts and $[\Sigma]$ totals below each bar. Inset tables summarize mapping and schema statistics (detailed in Appendix E).

5 Related Work

SHACL validation has been studied from multiple perspectives [7,20,25], including compilation into SQL to leverage relational database engines for validation [17] and exploiting shape constraints to optimize SPARQL query evaluation [26]. The data provenance semantics on which this work builds [10] formalizes shape fragments as subgraphs that witness conformance, but operates on materialized RDF graphs. Beyond provenance, shape fragments serve as a data retrieval mechanism related to triple pattern fragments [4,29]. No existing work encodes SHACL constraints into mapping rules, nor produces shape fragments at the mapping stage.

On the RML side, [22] provides a formal algebraic semantics for RML-Core, and [24] prunes RML mappings for a given SPARQL query by statically removing triples maps that cannot match any query triple pattern, but value-level constraints are not handled. OBDA systems such as Ontop [30] rewrite SPARQL queries into SQL over R2RML mappings under an injective template function assumption. RML2SHACL [11,13] derives SHACL shapes from RML mappings based on structural correspondences. The reverse direction, deriving an RML mapping from a SHACL schema alone, is not feasible, as a SHACL schema carries no information about source data or term generation. What is feasible is compiling SHACL constraints into an existing RML mapping, which is the approach taken in this work.

6 Conclusion and Future Work

We presented an approach that compiles SHACL shapes into existing RML mappings building on the provenance semantics for SHACL [10], producing

constraint-guided rewritten mappings that realize three progressively finer facets: relevance, conformance, and minimal deterministic evidence, with why-not provenance obtained through shape negation. We implemented an open-source system, SHACL2RML, proved and experimentally confirmed correctness of the approach, showing up to three orders of magnitude faster execution and lower memory usage than the post-hoc baseline. This work introduces the idea of coupling construction with validation, while the assumptions we make are the necessary foundation and starting point to establish this coupling within the current SHACL and RML models. The idea is bound neither to these assumptions nor to the mapping and validation languages that demonstrate it.

Our merge of the logical sources of multiple triples maps does not depend on the RML views [2,23] adopted in this work. Since the merge must retain all records from each source and align them by the subject they generate, any view mechanism supporting a full outer join can serve this role. The choice of RML views is a technological decision, and migrating to such mechanisms, for instance the actively evolving RML logical views [21], requires only replacing how the logical source of the merged triples map is constructed. This migration would also remove the lack of XML support in current RML views implementations.

To broaden the applicability of our implementation to any real-world schema, it automatically partitions the input schema into a supported schema for compilation and a deferred schema containing only unsupported constructs. For the excluded target node declaration, an intuitive workaround is to duplicate every triples map with a node equality function in every term map to identify the node generation, but this introduces redundancy proportional to the number of target nodes. Our assumptions and partially supported constructs are instead limited by the record-at-a-time model of RML engines, and together point to extending the engine’s processing model, on which the coupling can be carried further.

Supplemental Material Statement: Source code for SHACL2RML, all test cases, experimental scripts, raw results, and plotting scripts are available from <https://github.com/dtai-kg/SHACL2RML>. Full proofs of all lemmas and theorems are given in the appendix of this paper.

Acknowledgments. Xuemin Duan and Anastasia Dimou are partially supported by the European Union Horizon Research and Innovation programme under grant agreement No 101132389, REEVALUATE project. This research received funding from the Flemish Government under the “Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen” programme. David Chaves-Fraga acknowledges funding from the Spanish State Research Agency (Agencia Estatal de Investigación) through the REOPEN (PID2023-149549NB-I00). He also acknowledges the support of the Galician Ministry of Education, Universities and Vocational Training and the European Regional Development Fund (ERDF, “A way of making Europe”) through the “Centro de Investigación de Galicia accreditation (ED431G-2023/04, 2024-2027)” and the “Reference Competitive Group accreditation (ED431C 2026/52, 2026-2029)”. The collaboration of the authors is stimulated by the KG4DI FWO scientific research network (W001222N).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Declaration of use of Generative AI: Claude Code was utilized to assist in system implementation, experiment execution, and figure plot based on experiment results. All generated content was reviewed and verified by the authors.

References

1. Ahmetaj, S., Löhnert, B., Ortiz, M., Šimkus, M.: Magic shapes for SHACL validation. *Proceedings of the VLDB Endowment* **15**(10), 2284–2296 (2022)
2. Arenas-Guerrero, J., Alobaid, A., Navas-Loro, M., Pérez, M.S., Corcho, O.: Boosting knowledge graph generation from tabular data with RML views. In: *European Semantic Web Conference*. pp. 484–501. Springer (2023)
3. Arenas-Guerrero, Julián and Chaves-Fraga, David and Toledo, Jhon and Pérez, María S and Corcho, Oscar: Morph-KGC: Scalable knowledge graph materialization with mapping partitions. *Semantic Web* **15**(1), 1–20 (2024)
4. Berners-Lee, T.: Linked data shapes, forms and footprints. <https://www.w3.org/DesignIssues/Footprints.html> (2019), w3C Design Issues
5. Buneman, P., Khanna, S., Wang-Chiew, T.: Why and where: A characterization of data provenance. In: *International conference on database theory*. pp. 316–330. Springer (2001)
6. Chaves-Fraga, D., Priyatna, F., Cimmino, A., Toledo, J., Ruckhaus, E., Corcho, O.: GTFS-Madrid-Bench: A benchmark for virtual knowledge graph access in the transport domain. *Journal of Web Semantics* **65**, 100596 (2020). <https://doi.org/10.1016/j.websem.2020.100596>
7. Corman, J., Reutter, J.L., Savković, O.: Semantics and Validation of Recursive SHACL. In: *The Semantic Web – ISWC 2018. Lecture Notes in Computer Science*, vol. 11136, pp. 318–336. Springer (2018). https://doi.org/10.1007/978-3-030-00671-6_19
8. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF Mapping Language. Working group recommendation, World Wide Web Consortium (W3C) (Sep 2012), <http://www.w3.org/TR/r2rml/>
9. Delva, T.: pySHACL-fragments. <https://github.com/Shape-Fragments/pySHACL-fragments> (2023)
10. Delva, T., Dimou, A., Jakubowski, M., Van den Bussche, J.: Data provenance for SHACL. In: *Proceedings of the 26th International Conference on Extending Database Technology (EDBT)*. pp. 285–297. OpenProceedings.org (2023). <https://doi.org/10.48786/edbt.2023.23>
11. Delva, T., Smedt, B.D., Oo, S.M., Assche, D.V., Lieber, S., Dimou, A.: RML2shacl: RDF generation taking shape. In: *Proceedings of the 11th on Knowledge Capture Conference*. pp. 153–160. ACM (Dec 2021). <https://doi.org/10.1145/3460210.3493562>
12. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A generic language for integrated RDF mappings of heterogeneous data. In: *Proceedings of the 7th Workshop on Linked Data on the Web (LDOW)*. *CEUR Workshop Proceedings*, vol. 1184 (2014)
13. Duan, X., Chaves-Fraga, D., Derom, O., Dimou, A.: Scoop all the constraints’ flavours for your knowledge graph. In: *European Semantic Web Conference*. pp. 217–234. Springer (2024)
14. Elhalawati, A.A.A., Dimou, A., Van den Bussche, J.: A declarative formalization of r2rml using datalog and its efficient execution. In: *CEUR Workshop Proceedings*. R. Piskac c/o Redaktion Sun SITE Informatik V RWTH Aachen (2025)

15. Iglesias, E., Jozashoori, S., Chaves-Fraga, D., Collarana, D., Vidal, M.E.: SDM-RDFizer: An RML Interpreter for the Efficient Creation of RDF Knowledge Graphs. In: Proceedings of the 29th ACM International Conference on Information & Knowledge Management. ACM (Oct 2020). <https://doi.org/10.1145/3340531.3412881>
16. Iglesias-Molina, Ana and Van Assche, Dylan and Arenas-Guerrero, Julián and De Meester, Ben and Debruyne, Christophe and Jozashoori, Samaneh and Maria, Pano and Michel, Franck and Chaves-Fraga, David and Dimou, Anastasia: The RML ontology: A community-driven modular redesign after a decade of experience in mapping heterogeneous data to RDF. In: International Semantic Web Conference. pp. 152–175. Springer (2023)
17. Jakubowski, M., den Bussche, J.V.: Compiling shacl into sql. In: International Semantic Web Conference. pp. 59–77. Springer (2024)
18. Ke, J., Zacouris, Z., Acosta, M.: Efficient validation of shacl shapes with reasoning. Proceedings of the VLDB Endowment **17**(11), 3589–3601 (2024)
19. Knublauch, H., Kontokostas, D.: Shapes Constraint Language (SHACL). Recommendation, World Wide Web Consortium (W3C) (Jul 2017), <https://www.w3.org/TR/shacl/>
20. Leinberger, M., Seifer, P., Rienstra, T., Lämmel, R., Staab, S.: Deciding SHACL shape containment through description logics reasoning. In: International Semantic Web Conference. pp. 366–383. Springer (2020)
21. Maria, P., De Vleeschauwer, E., Lanti, D.: RML logical views. Draft Community Group Report, Knowledge Graph Construction Community Group (Jul 2026), <https://w3id.org/rml/lv/spec/>, accessed: 2026-08-05
22. Min Oo, S., Hartig, O.: An algebraic foundation for knowledge graph construction. In: European Semantic Web Conference. pp. 3–22. Springer (2025)
23. Morph-KGC Contributors: Morph-KGC documentation: RML views. <https://morph-kgc.readthedocs.io/en/stable/documentation/#rml-views> (2020), accessed: 2026-04-01
24. Oo, S.M., Hartig, O.: Query-Specific Pruning of RML Mappings. In: The Semantic Web – ISWC 2026. Lecture Notes in Computer Science, Springer (2026)
25. Pareti, P., Konstantinidis, G., Mogavero, F., Norman, T.J.: SHACL satisfiability and containment. In: International Semantic Web Conference. pp. 474–493. Springer (2020)
26. Thapa, R.B., Giese, M.: Optimizing sparql queries with shacl. In: International Semantic Web Conference. pp. 41–60. Springer (2023)
27. Van Assche, D., Chaves-Fraga, D., Dimou, A.: Krown: A benchmark for rdf graph materialisation. In: International semantic web conference. pp. 20–39. Springer (2024)
28. Van Nuffelen, B., Dekkers, M., Fragkou, P., De Cock, J., Schiltz, A., Sofou, A.: DCAT application profile for data portals in Europe (DCAT-AP) v3.0. SEMIC Recommendation, Interoperable Europe (2024), <https://semiceu.github.io/DCAT-AP/releases/3.0.0/>
29. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple pattern fragments: a low-cost knowledge graph interface for the web. Journal of Web Semantics **37**, 184–206 (2016)
30. Xiao, G., Lanti, D., Kontchakov, R., Komla-Ebri, S., Güzel-Kalaycı, E., Ding, L., Corman, J., Cogrel, B., Calvanese, D., Botoeva, E.: The virtual knowledge graph system ontop. In: International Semantic Web Conference. pp. 259–277. Springer (2020)

Table 4. Conformance $\mathcal{H}, G, a \models \varphi$, reproduced from [10, Table 1]. Following Remark 2, path expressions are restricted to direct predicate paths $p \in \mathcal{I}$ in our work.

φ	$\mathcal{H}, G, a \models \varphi$ iff
hasShape (s)	$\mathcal{H}, G, a \models \text{exp}(s, \mathcal{H})$
hasValue (c)	$a = c$
test (t)	a satisfies t (see text)
$\geq_n p.\varphi'$	$ \{b \in \llbracket p \rrbracket^G(a) \mid \mathcal{H}, G, b \models \varphi'\} \geq n$
$\leq_n p.\varphi'$	$ \{b \in \llbracket p \rrbracket^G(a) \mid \mathcal{H}, G, b \models \varphi'\} \leq n$
$\forall p.\varphi'$	$\mathcal{H}, G, b \models \varphi'$ for all $b \in \llbracket p \rrbracket^G(a)$
eq (p, p')	$\llbracket p \rrbracket^G(a) = \llbracket p' \rrbracket^G(a)$
disj (p, p')	$\llbracket p \rrbracket^G(a) \cap \llbracket p' \rrbracket^G(a) = \emptyset$
closed (P)	$p \in P$ for all $(a, p, b) \in G$
lessThan (p, p')	$b < c$ for all $b \in \llbracket p \rrbracket^G(a), c \in \llbracket p' \rrbracket^G(a)$
lessThanEq (p, p')	$b \leq c$ for all $b \in \llbracket p \rrbracket^G(a), c \in \llbracket p' \rrbracket^G(a)$
uniqueLang (p)	$b \not\sim c$ for all $b \neq c \in \llbracket p \rrbracket^G(a)$

Appendix A Data Provenance for SHACL

This appendix provides the formal definitions of conformance and neighborhood from [10]. The conformance semantics below defines when a node satisfies a shape expression, which our validation compilation (Section 3.3) encodes into RML mappings. The neighborhood and shape fragment definitions specify which triples constitute the provenance of each conforming node, which our neighborhood compilation aligns with.

A.1 Conformance Semantics

Given a schema $\mathcal{H}$, an RDF graph G , and a node $a \in \mathcal{N}$, the conformance relation $\mathcal{H}, G, a \models \varphi$ is defined inductively over the shape expression φ in Table 4, reproduced from [10, Table 1]. $\text{exp}(s, \mathcal{H})$ denotes the shape expression of s in $\mathcal{H}$, defaulting to $\top$. A path expression p evaluates on G to a binary relation $\llbracket p \rrbracket^G \subseteq \mathcal{N} \times \mathcal{N}$; we write $\llbracket p \rrbracket^G(a) = \{b \mid (a, b) \in \llbracket p \rrbracket^G\}$. Since this work is restricted to direct predicate paths (Remark 2), $\llbracket p \rrbracket^G(a) = \{b \mid (a, p, b) \in G\}$. For the boolean operators $\top$ (true), $\perp$ (false), $\neg$ (negation), $\wedge$ (conjunction), and $\vee$ (disjunction), the definition is standard. For node-level tests, $\mathcal{H}, G, a \models \text{test}(t)$ holds iff a satisfies $t \in \Omega$: **nodeKind**(k) requires a to have term type $k \in \{\text{sh:IRI}, \text{sh:Literal}, \text{sh:BlankNode}, \text{sh:BlankNodeOrIRI}, \text{sh:BlankNodeOrLiteral}, \text{sh:IRIOrLiteral}\}$; **datatype**(d) requires a to be a literal with datatype $d \in \mathcal{I}$; **langIn**(L) requires a to have a language tag in $L \subseteq \mathcal{L}$; **minIncl**(v) requires $a \geq v$, **maxIncl**(v) requires $a \leq v$, **minExcl**(v) requires $a > v$, **maxExcl**(v) requires $a < v$, where $v \in \mathcal{L}$ and $\geq, \leq, >, <$ follow the partial order on $\mathcal{L}$ from Definition 1; **minLen**(l) and **maxLen**(l) require $|a| \geq l$ and $|a| \leq l$ respectively, where $l \in \mathbb{N}$ and $|a|$ denotes the string length of a ; **pattern**(r) requires a to match regular expression r ; and **in**(S) requires $a \in S$.

Table 5. Neighborhood $B(v, G, \varphi)$ when $\mathcal{H}, G, v \models \varphi$ and φ is in NNF, reproduced from [10, Table 2]. Omitted cases yield the empty neighborhood. Path expressions are restricted to predicate paths $p \in \mathcal{I}$ (Remark 2) in our work.

φ	$B(v, G, \varphi)$
$\text{hasShape}(s)$	$B(v, G, \text{exp}(s, \mathcal{H}))$
$\neg \text{hasShape}(s)$	$B(v, G, \neg \text{exp}(s, \mathcal{H}))$
$\varphi_1 \wedge \varphi_2$	$B(v, G, \varphi_1) \cup B(v, G, \varphi_2)$
$\varphi_1 \vee \varphi_2$	$B(v, G, \varphi_1) \cup B(v, G, \varphi_2)$
$\geq_n p. \varphi'$	$\bigcup \{ \{(v, p, x)\} \cup B(x, G, \varphi') \mid (v, p, x) \in G, \mathcal{H}, G, x \models \varphi' \}$
$\leq_n p. \varphi'$	$\bigcup \{ \{(v, p, x)\} \cup B(x, G, \neg \varphi') \mid (v, p, x) \in G, \mathcal{H}, G, x \models \neg \varphi' \}$
$\forall p. \varphi'$	$\bigcup \{ \{(v, p, x)\} \cup B(x, G, \varphi') \mid (v, p, x) \in G \}$
$\text{eq}(p, p')$	$\{(v, p, x) \in G\} \cup \{(v, p', x) \in G\}$
$\neg \text{eq}(p, p')$	$\{(v, p, x) \in G \mid (v, p', x) \notin G\} \cup \{(v, p', x) \in G \mid (v, p, x) \notin G\}$
$\neg \text{disj}(p, p')$	$\bigcup \{ \{(v, p, x)\} \cup \{(v, p', x)\} \mid (v, p, x) \in G, (v, p', x) \in G \}$
$\neg \text{lessThan}(p, p')$	$\bigcup \{ \{(v, p, x)\} \cup \{(v, p', y)\} \mid (v, p, x) \in G, (v, p', y) \in G, x \not\prec y \}$
$\neg \text{lessThanEq}(p, p')$	$\bigcup \{ \{(v, p, x)\} \cup \{(v, p', y)\} \mid (v, p, x) \in G, (v, p', y) \in G, x \not\preceq y \}$
$\neg \text{uniqueLang}(p)$	$\{(v, p, x) \in G \mid \exists y \in \llbracket p \rrbracket^G(v) : y \neq x, y \sim x\}$
$\neg \text{closed}(P)$	$\{(v, p, x) \in G \mid p \notin P\}$

In our work, the inner shape expression φ' is restricted to path free constraints (Section 3.1), for which $B(x, G, \varphi') = B(x, G, \neg \varphi') = \emptyset$. The entries for $\geq_n$, $\leq_n$, and $\forall$ therefore simplify to $\bigcup \{ \{(v, p, x)\} \mid (v, p, x) \in G, \mathcal{H}, G, x \models \varphi' \}$, $\bigcup \{ \{(v, p, x)\} \mid (v, p, x) \in G, \mathcal{H}, G, x \models \neg \varphi' \}$, and $\bigcup \{ \{(v, p, x)\} \mid (v, p, x) \in G \}$, respectively.

A.2 Neighborhood and Shape Fragment

The *neighborhood* $B(v, G, \varphi)$ is defined inductively over φ following [10, Table 2], which is targeted by our work. Our validation compilation (Section 3.3) encodes conformance checks that determine whether a focus node produces triples, while our neighborhood compilation controls which triples each conforming node contributes. Since our work is restricted to direct predicate paths $p \in \mathcal{I}$ (Remark 2), the path constructs used in [10] simplify to: $\text{graph}(\text{paths}(p, G, v, x)) = \{(v, p, x)\}$ and $\text{pathsfrom}(p, G, v) = \{(v, p, x) \mid (v, p, x) \in G\}$. Table 5 defines $B(v, G, \varphi)$ when $\mathcal{H}, G, v \models \varphi$ and φ is in NNF. If $\mathcal{H}, G, v \not\models \varphi$ or φ is among the omitted cases ($\top$, $\perp$, $\text{hasValue}(c)$, $\text{test}(t)$, $\text{disj}(p, p')$, $\text{lessThan}(p, p')$, $\text{lessThanEq}(p, p')$, $\text{uniqueLang}(p)$, $\text{closed}(P)$, $\neg \text{hasValue}(c)$, $\neg \text{test}(t)$), the neighborhood is empty. The neighborhood is designed to be deterministic and sufficient [10], and as minimal as possible; however, minimality is not always achievable together with determinism and sufficiency [10, Remark 3.6].

The *shape fragment* definition (Definition 5) uses the schema form $\text{Frag}(G, \mathcal{H})$, a special case of the general form $\text{Frag}(G, S) = \bigcup \{ B(v, G, \varphi) \mid v \in \mathcal{N}, \varphi \in S \}$ defined over arbitrary shape sets S in [10]. The schema form derives S from $\mathcal{H}$ by conjoining each shape expression with its target ($S = \{ \varphi \wedge \tau \mid \exists s : (s, \varphi, \tau) \in \mathcal{H} \}$), ensuring that only targeted nodes contribute and that target-witnessing triples are included in the neighborhood. This work uses the schema form because the

general form requires enumerating all nodes without target declarations, which cannot be aligned to mapping rules statically (cf. Remark 1).

Appendix B Merged Logical Source Example

Following Example 4, the `sqljoin` for $\mathcal{A}'(\text{ArticleShape}, \varphi_1, \tau_1)$ produces the following SQL query as the logical source of the merged triples map, where $\text{sid}(sm)$ translates a subject map into a SQL expression, a column name for `ref` and a string concatenation for `tmpl`, and σ_i renames the references of tm_i :

```
SELECT COALESCE(t1.__sid__, t2.__sid__) AS __sid__, t1_year, t2_year
FROM (SELECT 'http://article.org/'
      || url_encode(CAST(id AS VARCHAR)) || '_'
      || url_encode(CAST(title AS VARCHAR)) AS __sid__,
      year AS t1_year FROM 'article.csv') AS t1
FULL OUTER JOIN (SELECT 'http://article.org/'
      || url_encode(CAST(doi AS VARCHAR)) || '_'
      || url_encode(CAST(title AS VARCHAR)) AS __sid__,
      year AS t2_year FROM 'journal.csv') AS t2
ON t1.__sid__ = t2.__sid__
```

Appendix C Constraint Function Semantics

This appendix gives the evaluation semantics of every constraint function f in Tables 1, 2, and 3, together with their encoding as `fexec` term maps.

Definition 16 (Logical Constraint Functions). For constraint functions f_1, f_2, f and a record r , following the conformance semantics of [10] (Appendix A.1):

$$\begin{aligned} \text{eval}(f_{\text{and}}(f_1, f_2), r) &= \begin{cases} \text{none} & \text{if } \text{eval}(f_1, r) = \text{eval}(f_2, r) = \text{none} \\ \text{false} & \text{if } \text{eval}(f_1, r) = \text{false} \text{ or } \text{eval}(f_2, r) = \text{false} \\ \text{true} & \text{otherwise} \end{cases} \\ \text{eval}(f_{\text{or}}(f_1, f_2), r) &= \begin{cases} \text{none} & \text{if } \text{eval}(f_1, r) = \text{eval}(f_2, r) = \text{none} \\ \text{false} & \text{if } \text{eval}(f_1, r) = \text{eval}(f_2, r) = \text{false} \\ \text{true} & \text{otherwise} \end{cases} \\ \text{eval}(f_{\text{not}}(f), r) &= \begin{cases} \text{true} & \text{if } \text{eval}(f, r) = \text{false} \\ \text{false} & \text{if } \text{eval}(f, r) = \text{true} \\ \text{none} & \text{if } \text{eval}(f, r) = \text{none} \end{cases} \end{aligned}$$

Definition 17 (Atomic Constraint Functions). For a term map map and a test $t \in \Omega_V$ other than `in`, with parameter v :

$$\text{eval}(f_t(map, \text{const}(v)), r) = \begin{cases} \text{true} & \text{if } \text{eval}(map, r) \neq \text{none} \text{ and } \mathcal{H}, G, \text{eval}(map, r) \models \text{test}(t(v)) \\ \text{false} & \text{if } \text{eval}(map, r) \neq \text{none} \text{ and } \mathcal{H}, G, \text{eval}(map, r) \not\models \text{test}(t(v)) \\ \text{none} & \text{if } \text{eval}(map, r) = \text{none} \end{cases}$$

where conformance $\models$ is defined in Appendix A.1. For a `hasValue(c)`:

$$\text{eval}(f_{\text{eq}}(\text{map}, \text{const}(c)), r) = \begin{cases} \text{true} & \text{if } \text{eval}(\text{map}, r) = c \\ \text{false} & \text{if } \text{eval}(\text{map}, r) \neq c \text{ and } \text{eval}(\text{map}, r) \neq \text{none} \\ \text{none} & \text{if } \text{eval}(\text{map}, r) = \text{none} \end{cases}$$

For `in(S)`, whose parameter is a finite set of constants S :

$$\text{eval}(f_{\text{in}}(\text{map}, \text{const}(S)), r) = \begin{cases} \text{true} & \text{if } \text{eval}(\text{map}, r) \in S \\ \text{false} & \text{if } \text{eval}(\text{map}, r) \notin S \text{ and } \text{eval}(\text{map}, r) \neq \text{none} \\ \text{none} & \text{if } \text{eval}(\text{map}, r) = \text{none} \end{cases}$$

Definition 18 (Set-Level Constraint Functions). Let $(\cdot, om_1), \dots, (\cdot, om_k) \in \text{poms}_p(tm)$ and $(\cdot, om'_1), \dots, (\cdot, om'_l) \in \text{poms}_{p'}(tm)$. Define the *value sets* as $S_p(r) = \{\text{eval}(om_i, r) \mid i \in [k]\} \setminus \{\text{none}\}$ and $S_{p'}(r) = \{\text{eval}(om'_j, r) \mid j \in [l]\} \setminus \{\text{none}\}$. And we have:

$$\begin{aligned} \text{eval}(f_{\text{eq}}(\bar{om}_p, \bar{om}_{p'}), r) &= [S_p(r) = S_{p'}(r)] \\ \text{eval}(f_{\text{disj}}(\bar{om}_p, \bar{om}_{p'}), r) &= [S_p(r) \cap S_{p'}(r) = \emptyset] \\ \text{eval}(f_{\text{lt}}(\bar{om}_p, \bar{om}_{p'}), r) &= [\forall s \in S_p(r), \forall s' \in S_{p'}(r) : s < s'] \\ \text{eval}(f_{\text{leq}}(\bar{om}_p, \bar{om}_{p'}), r) &= [\forall s \in S_p(r), \forall s' \in S_{p'}(r) : s \leq s'] \end{aligned}$$

Following the *neighborhood* definition of [10] (Appendix A.2), the negated forms in Table 3 use existential variants comparing one object map against a value set. For an object map om with $x = \text{eval}(om, r)$, a sequence $\bar{om}_q$ with value set $S_q(r)$, and $\bowtie \in \{=, <, \neq, \leq, \geq\}$ with the corresponding $f_{\text{eq}}^\bowtie, f_{\text{geq}}^\bowtie, f_{\text{leq}}^\bowtie, f_{\text{gt}}^\bowtie, f_{\text{lt}}^\bowtie$:

$$\text{eval}(f_{\bowtie}^\bowtie(om, \bar{om}_q), r) = [\exists s \in S_q(r) : x \bowtie s]$$

where $[\cdot]$ denotes `true` if the condition holds, `false` otherwise, and each $f^\bowtie$ evaluates to `none` if $x = \text{none}$. Since $<$ is a strict partial order (Definition 1), the neighborhood conditions of [10] for `¬lessThan` and `¬lessThanEq` require the complements $x \not< y$ and $x \not\leq y$ (cf. Table 5) rather than $x \geq y$ and $x > y$; the two coincide only on comparable values, and the function names reflect the total-order reading. No existential variant of f_{disj} is needed: with a single object map as its first argument, $f_{\text{disj}}(om, \bar{om}_{p'})$ already evaluates to $[x \notin S_{p'}(r)]$, the counterexample condition for `eq`. A single object map is thus read as a one-element sequence throughout. The name f_{eq} is used for both the comparison of a term map against a constant (Definition 17) and the comparison of two value sets above; the arguments determine which applies.

Definition 19 (Counting Function). For a finite set of pairs $\{(om_i, \varepsilon_i)\}_{i \in [k]}$, where each om_i is an object map and each ε_i is a rewriting decision (Definition 13), and an integer bound n :

$$\begin{aligned} \text{eval}(f_{\text{count} \geq}(\{(om_i, \varepsilon_i)\}_{i \in [k]}, n), r) &= \\ &[\left| \{\text{eval}(om_i, r) \mid i \in [k], \text{eval}(\varepsilon_i, r) \neq \text{false}, \text{eval}(om_i, r) \neq \text{none}\} \right| \geq n] \\ \text{eval}(f_{\text{count} \leq}(\{(om_i, \varepsilon_i)\}_{i \in [k]}, n), r) &= \\ &[\left| \{\text{eval}(om_i, r) \mid i \in [k], \text{eval}(\varepsilon_i, r) \neq \text{false}, \text{eval}(om_i, r) \neq \text{none}\} \right| \leq n] \end{aligned}$$

where $\varepsilon_i = \top$ is treated as $\text{eval}(\varepsilon_i, r) = \text{true}$ for all r , and $\varepsilon_i = \perp$ as $\text{eval}(\varepsilon_i, r) = \text{false}$ for all r . The cardinality is taken over the *set* of values rather than over the object maps: distinct object maps may evaluate to the same value on a record, while conformance (Appendix A.1) counts the distinct values in $\llbracket p \rrbracket^G(a)$; since φ' within $\#_n^m p.\varphi'$ is path free, its truth is determined by the value alone, so object maps evaluating to the same value yield the same outcome for ε_i .

Definition 20 (Conditional Suppression Function). For a term map $tmap$ and a constraint function f :

$$\text{eval}(f_{\text{passif}}(tmap, f), r) = \begin{cases} \text{eval}(tmap, r) & \text{if } \text{eval}(f, r) \in \{\text{true}, \text{none}\} \\ \text{none} & \text{otherwise} \end{cases}$$

Since $f_{\text{passif}}(tmap, f)$ returns $\text{eval}(tmap, r)$ or none , it declares the same term type, datatype, and language tag as $tmap$.

Definition 21 (Function Encoding). Each constraint function $f(\arg_1, \dots, \arg_k)$ is encoded as an fexec term map (Definition 7):

$$f(\arg_1, \dots, \arg_k) = \text{fexec}((f_{\text{IRI}}, \{(par_1, \arg_1), \dots, (par_k, \arg_k)\}))$$

where $f_{\text{IRI}} \in \mathcal{I}$ is the FnO function IRI corresponding to f , and each $par_i \in \mathcal{I}$ is a parameter IRI. The return map ρ is omitted throughout as all functions produce a single output. The specific encodings are:

- $f_{\text{and}}(f_1, f_2) = \text{fexec}((:\text{and}, \{(:\text{par}_1, f_1), (:\text{par}_2, f_2)\}))$
- f_{or} and f_{not} analogously with IRIs $:\text{or}$ and $:\text{not}$
- $f_t(map, \text{const}(v)) = \text{fexec}((f_t, \{(:\text{par}_v, map), (:\text{par}_n, \text{const}(v))\}))$ where f_t corresponds to the constraint (cf. Table 1)
- $f_{\text{eq}}(om_p, om_{p'}) = \text{fexec}((:\text{equals_set}, \{(:\text{left}_1, om_1), \dots, (:\text{left}_k, om_k), (:\text{right}_1, om'_1), \dots, (:\text{right}_l, om'_l)\}))$; f_{disj} , f_{lt} , f_{leq} , and the existential variants $f_{\text{eq}}^\exists$, $f_{\text{geq}}^\exists$, $f_{\text{gt}}^\exists$, $f_{\text{leq}}^\exists$, $f_{\text{lt}}^\exists$ analogously
- $f_{\text{count}_{\geq}}(\{(om_i, \varepsilon_i)\}, n) = \text{fexec}((:\text{countGeq}, \{(:\text{val}_1, f_{\text{passif}}(om_1, \varepsilon_1)), \dots, (:\text{val}_k, f_{\text{passif}}(om_k, \varepsilon_k)), (:\text{par}_n, \text{const}(n))\}))$; $f_{\text{count}_{\leq}}$ analogously with $:\text{countLeq}$
- $f_{\text{passif}}(tmap, f) = \text{fexec}((:\text{passif}, \{(:\text{par}_v, tmap), (:\text{par}_f, f)\}))$

Appendix D Full Proofs

Throughout this appendix, given a schema $\mathcal{H}$ and an RML mapping Σ , let $G = \llbracket \Sigma \rrbracket$, Σ'_{RE} be the output of $\text{rewrite}(\Sigma, \mathcal{H}, \text{RE})$, $\Pi = \text{consolidate}(\Sigma, \mathcal{H}, \text{CO})$ (equivalently $\Pi = \text{consolidate}(\Sigma, \mathcal{H}, \text{EV})$), and Σ'_{CO} , Σ'_{EV} be the outputs of $\text{rewrite}(\Sigma, \mathcal{H}, \text{CO})$ and $\text{rewrite}(\Sigma, \mathcal{H}, \text{EV})$, respectively. $\llbracket \Sigma'_{\text{mode}}(s, \varphi, \tau) \rrbracket$ denotes the triples generated by the rewritten triples map of (s, φ, τ) in Σ'_{mode} , and $\emptyset$ when $(s, \varphi, \tau) \notin \text{dom}(\Pi)$ or the triples map is discarded by Algorithm 2. For a focus node v of (s, φ, τ) , we write $map_v \in \text{fmaps}(tm, \tau)$ for a term map generating v and r_v for the record on which it does so. For $\tau = \geq_1 p_\tau^- . \top$ several such pairs (map_v, r_v) may generate v , and the claims below hold for each of them.

D.1 Proof of Lemma 1 (Consolidation Completeness)

Proof. Let $(s, \varphi, \tau) \in \mathcal{H}$ and let v with $G, v \models \tau$. If $\text{align}((s, \varphi, \tau), \Sigma) = \emptyset$, then no such v exists and the claim holds vacuously: for $\tau = \neg \top$ no node satisfies τ ; for $\tau = \geq_1 \text{rdf:type.hasValue}(c)$ ($\geq_1 \text{a.hasValue}(c)$) a focus node requires $(v, \mathbf{a}, c) \in G$, whose generating triples map has a subject map generating an instance of c and hence declares $(\text{const}(\mathbf{a}), \text{const}(c))$ by Remark 5, so it would be aligned; for $\tau = \geq_1 p_\tau. \top$ and $\tau = \geq_1 p_\tau^- . \top$ a focus node requires a p_τ -triple in G , whose generating triples map has a *pom* with predicate map $\text{const}(p_\tau)$ by Remark 4, so it would be aligned. We henceforth assume $\text{align}((s, \varphi, \tau), \Sigma) \neq \emptyset$. A relevant triple of v is either a triple (v, p, o) with $p \in P(\varphi)$, or a triple witnessing τ .

Target-witnessing triples. For $\tau = \geq_1 \text{a.hasValue}(c)$ the witness is $(v, \mathbf{a}, c)$, generated by a *pom* $(\text{const}(\mathbf{a}), \text{const}(c))$; every triples map carrying such a *pom* is aligned by Definition 10 case 1 and the *pom* is retained by Definition 11 case 1. For $\tau = \geq_1 p_\tau. \top$ the witness is (v, p_τ, o) and for $\tau = \geq_1 p_\tau^- . \top$ it is (s', p_τ, v) ; in both cases the witness carries predicate p_τ , so the triples map generating it has a *pom* with predicate p_τ , is aligned by Definition 10 cases 2 and 3, and the *pom* is retained by Definition 11 cases 2 and 3.

Triples with $p \in P(\varphi)$. Under $\tau = \geq_1 p_\tau. \top$ and $\tau = \geq_1 p_\tau^- . \top$, φ is restricted to constraints on the focus node itself (Section 3.1), which mention no predicate, so $P(\varphi) = \emptyset$ and the witness above is the only relevant triple. For $\tau = \geq_1 \text{a.hasValue}(c)$, let $(v, p, o) \in G$ with $p \in P(\varphi)$. There exist $tm \in \Sigma$ and a record r of tm with $\text{eval}(\text{sm}(tm), r) = v$ such that (v, p, o) is generated by some $(pm, om) \in \text{poms}(tm)$ with $pm = \text{const}(p)$. We show that tm is aligned: since $G, v \models \tau$, $(v, \mathbf{a}, c) \in G$, and by Remark 5 every triples map whose subject map generates v declares class c , so tm is aligned by Definition 10 case 1. Since $p \in P(\varphi)$, the *pom* (pm, om) is retained by pruning (Definition 11), so $(v, p, o) \in \llbracket tm' \rrbracket$ where tm' is the pruned copy.

Let tm' denote the aligned and pruned copy of tm , so that every relevant triple of v is in $\llbracket tm' \rrbracket$. For $\Sigma'_{\text{RE}}(s, \varphi, \tau)$, which is the set of all such pruned triples maps (Algorithm 1), the triple is therefore in $\llbracket \Sigma'_{\text{RE}}(s, \varphi, \tau) \rrbracket$. For $\Pi(s, \varphi, \tau)$: if only one triples map is aligned, $\Pi(s, \varphi, \tau) = tm'$ directly. If multiple triples maps are aligned, merging via FULL OUTER JOIN on `--sid--` preserves all records: by injectivity (Remark 4), the record generating v produces a unique `--sid--` value equal to v 's IRI, so it appears in the joined virtual table, and the renamed *pom* still generates the triple. Thus every relevant triple of v is in $\llbracket \Pi(s, \varphi, \tau) \rrbracket$.

D.2 Proof of Lemma 2 (Consolidation Soundness)

Proof. Let $(s, \varphi, \tau) \in \mathcal{H}$ and let (s', p, o) be a triple generated by $\Sigma'_{\text{RE}}(s, \varphi, \tau)$ or by $\Pi(s, \varphi, \tau)$ that introduces no new triples. The triple (s', p, o) is generated by a pruned, aligned copy of some $tm \in \Sigma$; since pruning only removes *poms* and merging only renames references over the joined records, $(s', p, o) \in \llbracket tm \rrbracket \subseteq G$.

By Definition 10, tm was aligned because it matches τ . For $\tau = \geq_1 \text{a.hasValue}(c)$, tm declares class c , so each of its records generating a subject also generates $(s', \mathbf{a}, c)$, hence $G, s' \models \tau$; the triple (s', p, o) is a relevant triple of s' since, by

Definition 11 case 1, $p \in P(\varphi)$ or $p = \mathbf{a}$ with $o = c$ witnessing τ . For $\tau = \geq_1 p_\tau \cdot \top$, by Definition 11 case 2 the retained *poms* carry predicate p_τ (as $P(\varphi) = \emptyset$, Section 3.1), so $p = p_\tau$ and (s', p_τ, o) itself witnesses $G, s' \models \tau$; it is a relevant triple of s' . For $\tau = \geq_1 p_\tau^- \cdot \top$, by Definition 11 case 3 the retained *poms* carry predicate p_τ , so $p = p_\tau$ and the triple (s', p_τ, o) witnesses $G, o \models \tau$; it is a target-witnessing, and hence relevant, triple of the focus node o . In each case the generated triple is a relevant triple of a node v with $G, v \models \tau$.

D.3 Proof of Lemma 3 (Validation Compilation Completeness)

Proof. Let $(s, \varphi, \tau) \in \mathcal{H}$, let $tm = \Pi(s, \varphi, \tau)$, and let v with $G, v \models \tau$ and $G, v \models \varphi$. By Lemma 1, every relevant triple of v is in $\llbracket \Pi(s, \varphi, \tau) \rrbracket$.

When $map_v = \mathbf{sm}(tm)$, injectivity (Remark 4) makes r_v the unique record of $\mathbf{ls}(tm)$ with $\mathbf{eval}(\mathbf{sm}(tm), r_v) = v$, and by Lemmas 1 and 2, for every predicate $q \in P(\varphi)$, $\{\mathbf{eval}(om, r_v) \mid (\cdot, om) \in \mathbf{poms}_q(tm)\} \setminus \{\mathbf{none}\} = \llbracket q \rrbracket^G(v)$, where $\llbracket q \rrbracket^G(a) = \{b \mid (a, q, b) \in G\}$ denotes the values of a along p in G . Indeed, every q -triple of v is a relevant triple, so it is generated by tm , necessarily on r_v and by some retained *pom* with predicate q ; conversely, every value generated by such a *pom* on r_v yields a triple of $\llbracket tm \rrbracket \subseteq G$ with subject v and predicate q .

Writing $\varepsilon = \mathbf{compile}_v(\psi, map, tm, \mathcal{H})$, we prove the following two claims by simultaneous induction on the structure of shape expressions, for every sub-expression ψ of φ (after unfolding referenced shapes via $\mathbf{exp}(\cdot, \mathcal{H})$) and every term map map among $\mathbf{fmaps}(tm, \tau)$ and the object maps of tm , on any record r_v where $x = \mathbf{eval}(map, r_v) \neq \mathbf{none}$:

- (a) if $\mathcal{H}, G, x \models \psi$, then $\varepsilon \neq \perp$ and, if ε is some f , $\mathbf{eval}(f, r_v) \in \{\mathbf{true}, \mathbf{none}\}$;
- (b) if $\mathcal{H}, G, x \not\models \psi$, then $\varepsilon \neq \top$ and, if ε is some f , $\mathbf{eval}(f, r_v) = \mathbf{false}$.

Since the constructs `eq`, `disj`, `lessThan`, `lessThanEq`, `uniqueLang`, `closed`, $\forall$, and $\#_n^m$ reference predicates of the focus node, they occur only at the focus-node level, with $map = map_v$ and $x = v$; within $\forall p.\varphi'$ and $\#_n^m p.\varphi'$, the path-free φ' (Section 3.1) is a Boolean combination of `test` and `hasValue`, whose compiled functions evaluate to `none` exactly when the object map evaluates to `none` (Appendix C), and to `true` or `false`, determined by the value alone, otherwise.

Constants and node tests. For $\top$ and $\perp$ the claims are immediate. For `test`(t) with $t \in \Omega_T$: $\mathbf{compile}_v$ statically checks the term type, datatype, or language tag that map declares, defaults to, or infers, which determines the corresponding property of every value it yields, in particular of x . Hence the declaration is compatible with t iff $x \models \mathbf{test}(t)$, giving (a) and (b). For `test`($t(w)$) with $t \in \Omega_V$: $\varepsilon = f_t(map, \mathbf{const}(w))$; since $x \neq \mathbf{none}$, f_t evaluates to `true` iff x satisfies $t(w)$ and to `false` otherwise (Definition 17), giving (a) and (b). If all arguments are constants, the static folding of the same value yields the same result. For `hasValue`(c): analogous with $f_{\mathbf{eq}}(map, \mathbf{const}(c))$ and the condition $x = c$.

Predicate comparisons. For `eq`(p, p'), `disj`(p, p'), `lessThan`(p, p'), `lessThanEq`(p, p'): ε is the set-level function on $\tilde{o}m_p$ and $\tilde{o}m_{p'}$. Since $p, p' \in P(\varphi)$, the value sets $S_p(r_v)$ and $S_{p'}(r_v)$ (Definition 18) equal $\llbracket p \rrbracket^G(v)$ and $\llbracket p' \rrbracket^G(v)$, so the evaluation

conditions coincide with the conformance conditions of Table 4, and the function evaluates to **true** iff $v \models \psi$ and to **false** otherwise, giving (a) and (b). For $\text{uniqueLang}(p)$: the decision is static. For (b), $v \not\models \text{uniqueLang}(p)$ means two distinct p -values of v carry the same language tag; both are values of object maps in $\text{poms}_p(tm)$, and an object map's values carry a tag exactly when it declares $\text{lang}(\ell)$, so two object maps declare the same tag and $\varepsilon = \perp \neq \top$. For (a), suppose $\varepsilon = \perp$, i.e., two object maps declare the same tag ℓ ; if both yield values on r_v , these are two distinct values tagged ℓ , contradicting $v \models \text{uniqueLang}(p)$; the remaining case, in which one of them yields no value on r_v , is the degenerate case excluded in Section 3.3. Hence $\varepsilon = \top$, and (a) holds. For $\text{closed}(P)$: the decision is static, and the construct occurs only under $\tau = \geq_1 \mathbf{a}.\text{hasValue}(c)$ by Section 3.1. For (b), $v \not\models \text{closed}(P)$ gives a triple $(v, q, o) \in G$ with $q \notin P$; by Remark 5 its generating triples map is aligned and merged into tm , so $q \in \text{preds_orig}(tm)$ and $\varepsilon = \perp \neq \top$. For (a), suppose $\varepsilon = \perp$, i.e., some $q \in \text{preds_orig}(tm) \setminus P$; since $v \models \text{closed}(P)$, no pom with predicate q yields a value on r_v , which is again the degenerate case excluded; hence $\varepsilon = \top$.

Boolean combinations. For $\varphi_1 \wedge \varphi_2$, (a): both conjuncts are satisfied, so by induction neither sub-decision is $\perp$ and runtime sub-decisions do not evaluate to **false**; by the $\hat{\wedge}$ reduction (Definition 13) $\varepsilon \neq \perp$, and f_{and} evaluates to **false** only if a constituent does (Appendix C). (b): some conjunct is violated, so by induction its sub-decision is $\perp$, making $\varepsilon = \perp$, or some f evaluating to **false**, making ε either that f or an f_{and} dominated by the **false** constituent; in all cases $\varepsilon \neq \top$ and the evaluation is **false**. For $\varphi_1 \vee \varphi_2$, (a): some disjunct is satisfied; by induction its sub-decision is $\top$, making $\varepsilon = \top$, or some f with evaluation in $\{\text{true}, \text{none}\}$; by the $\hat{\vee}$ reduction $\varepsilon \neq \perp$, and f_{or} evaluates to **false** only if both constituents do (Appendix C). (b): both disjuncts are violated, so by induction each sub-decision is $\perp$ or evaluates to **false**; hence $\varepsilon = \perp$ or a runtime check evaluating to **false**, and $\varepsilon \neq \top$. For $\text{hasShape}(s')$: compile_v unfolds to $\text{exp}(s', \mathcal{H})$ and the claims follow by induction. For $\neg\psi'$: by the NNF assumption (Definition 4) ψ' is atomic. If ψ' is atomic and not hasShape , then by the base cases above its compiled runtime check never evaluates to **none** when $x \neq \text{none}$; claims (a) and (b) for $\neg\psi'$ then follow from (b) and (a) for ψ' via the $\hat{\neg}$ reduction and f_{not} (Appendix C). If $\psi' = \text{hasShape}(s')$, the same argument applies to $\text{exp}(s', \mathcal{H})$, except that in case (b) the inner evaluation may be **none**; this occurs only when every object map referenced by the unfolded expression yields no value on r_v , i.e., when $\text{exp}(s', \mathcal{H})$ holds vacuously because the relevant triples are absent, which is the degenerate case excluded; hence the inner evaluation is **true** and f_{not} evaluates to **false**.

Path quantifiers. For $\forall p.\varphi'$: if $\text{poms}_p(tm) = \emptyset$, then $\varepsilon = \top$; for (b), note that $p \in P(\varphi)$, so $\llbracket p \rrbracket^G(v) = \emptyset$ and $v \models \forall p.\varphi'$ vacuously, making (b) vacuous. Otherwise, (a): every p -value of v satisfies φ' . For each $(\cdot, om) \in \text{poms}_p$ with $\text{eval}(om, r_v) \neq \text{none}$, the value is in $\llbracket p \rrbracket^G(v)$ and satisfies φ' , so by induction its sub-decision is not $\perp$ and does not evaluate to **false**; a sub-decision $\perp$ could thus stem only from an object map yielding no value on r_v while v conforms, which is the degenerate case excluded. Hence $\varepsilon \neq \perp$, and the f_{and} chain evaluates over constituents in $\{\text{true}, \text{none}\}$, absorbing **none** (Appendix C), so it does not evaluate

to false. (b): some p -value $b = \text{eval}(om, r_v) \neq \text{none}$ violates φ' ; by induction its sub-decision is $\perp$, making $\varepsilon = \perp$, or evaluates to false, dominating the f_{and} chain; in all cases $\varepsilon \neq \top$ and the evaluation is false. For $\geq_n p.\varphi'$ with $\text{poms}_p \neq \emptyset$: $\varepsilon = f_{\text{count}_{\geq}}(\Phi_p, n)$ with Φ_p as in Table 2. By induction on the path-free φ' , the qualifying value set of Definition 19 equals $\{b \in \llbracket p \rrbracket^G(v) \mid \mathcal{H}, G, b \models \varphi'\}$; every qualifying value is a p -value of v whose sub-decision is $\top$, or a runtime check evaluating to true, hence a value satisfying φ' by the contrapositive of (b); conversely every p -value satisfying φ' is the value of some object map in poms_p on r_v whose sub-decision qualifies by (a). Therefore $f_{\text{count}_{\geq}}$ evaluates to true iff $v \models \geq_n p.\varphi'$ (Table 4), giving (a) and (b). If $\text{poms}_p = \emptyset$: for $n = 0$, $\varepsilon = \top$ and v conforms trivially; for $n \geq 1$, $\varepsilon = \perp$, and $\llbracket p \rrbracket^G(v) = \emptyset$, so $v \not\models \geq_n p.\varphi'$ and (a) is vacuous. For $\leq_n p.\varphi'$: analogous with $f_{\text{count}_{\leq}}$; when $\text{poms}_p = \emptyset$, $\varepsilon = \top$ and v conforms vacuously.

Instantiating (a) with $\psi = \varphi$, $\text{map} = \text{map}_v$, and $x = v$: since $G, v \models \varphi$, $\varepsilon_v \neq \perp$, so map_v is not dropped (Algorithm 2). If $\varepsilon_v = \top$, map_v is unchanged. If ε_v is some f , then $\text{eval}(f, r_v) \neq \text{false}$, so $f_{\text{passif}}(\text{map}_v, f)$ returns $\text{eval}(\text{map}_v, r_v) = v$ (Definition 20). For $\tau = \geq_1 p_{\tau}^{-}.\top$ this holds for each pair (map_v, r_v) generating v . In either case no relevant triple of v is removed, and by Lemma 1 all of them are generated by tm . Thus every relevant triple of v is in $\llbracket \Sigma'_{\text{CO}}(s, \varphi, \tau) \rrbracket$.

D.4 Proof of Lemma 4 (Validation Compilation Soundness)

Proof. Let $(s, \varphi, \tau) \in \mathcal{H}$ and let v with $G, v \models \tau$ and $G, v \not\models \varphi$. Instantiating claim (b), established jointly with claim (a) in the proof of Lemma 3, with $\psi = \varphi$, $\text{map} = \text{map}_v$, and $x = v \neq \text{none}$: either $\varepsilon_v = \perp$, or ε_v is some f with $\text{eval}(f, r_v) = \text{false}$. If $\varepsilon_v = \perp$, then $\text{drop}(tm, \text{map}_v)$ is applied (Algorithm 2), which removes map_v from tm together with its predicate-object map when map_v is an object map, and discards tm when map_v is its subject map. If ε_v is some f , then $f_{\text{passif}}(\text{map}_v, f)$ returns none on r_v (Definition 20), so map_v generates no term on that record. In both cases map_v yields no term on r_v . Every relevant triple of v requires that term: when $\text{map}_v = \text{sm}(tm)$, each such triple has v as its subject, is generated only on r_v (injectivity, Remark 4), and only if the subject map yields v ; when $\tau = \geq_1 p_{\tau}^{-}.\top$, the relevant triples of v are exactly the target-witnessing triples carrying v as object (Lemma 1), each generated by the predicate-object map of one pair (map_v, r_v) , and since ε_v depends on the value v alone (φ consists of node tests, Section 3.1), every such pair is suppressed in the same way. Hence no relevant triple of v is in $\llbracket \Sigma'_{\text{CO}}(s, \varphi, \tau) \rrbracket$.

D.5 Proof of Lemma 5 (Neighborhood Compilation Completeness)

Proof. Let $(s, \varphi, \tau) \in \mathcal{H}$ and let v with $G, v \models \tau$ and $G, v \models \varphi$. By Lemma 3, all relevant triples of v are in $\llbracket \Sigma'_{\text{CO}}(s, \varphi, \tau) \rrbracket$; in particular, the validation injections on $\text{fmaps}(tm, \tau)$ pass on the records generating v . By Algorithm 2, compile_n is invoked on $\varphi \wedge \tau$; by the $\wedge$ entry of Table 3, its result is the union of $\text{compile}_n(\varphi, \cdot)$ and $\text{compile}_n(\tau, \cdot)$, and correspondingly, since $v \models \varphi$ and $v \models \tau$,

$B(v, G, \varphi \wedge \tau) = B(v, G, \varphi) \cup B(v, G, \tau)$ (Table 5). We show by structural induction on φ , and then for τ , that every triple of $B(v, G, \varphi \wedge \tau)$ is generated. Since φ' within $\forall p.\varphi'$ or $\#_n^m p.\varphi'$ is path free (Section 3.1), i.e., a Boolean combination of `test` and `hasValue` whose neighborhoods and negated neighborhoods are empty, $B(x, G, \varphi') = B(x, G, \neg\varphi') = \emptyset$ for all reached nodes x , simplifying the neighborhood entries. Throughout, claims (a) and (b) refer to the simultaneous induction established in the proof of Lemma 3, and they apply unchanged to a f_{passif} -wrapped term map (Definition 20).

Path quantifiers. For $\forall p.\varphi'$: B reduces to $\{(v, p, x) \mid (v, p, x) \in G\}$, and compile_n retains poms_p unfiltered; these poms generate every p -triple of v on r_v . For $\geq_n p.\varphi'$: B reduces to $\{(v, p, x) \mid (v, p, x) \in G, G, x \models \varphi'\}$, and compile_n applies resolve_n with $\varepsilon_{om} = \text{compile}_v(\varphi', om, tm, \mathcal{H})$. For an object x satisfying φ' , ε_{om} is $\top$ or evaluates to `true` on r_v by claim (a), so the pom is retained, unfiltered or with a filter that passes. For $\leq_n p.\varphi'$: B reduces to $\{(v, p, x) \mid (v, p, x) \in G, G, x \not\models \varphi'\}$, and compile_n applies resolve_n with $\hat{\varepsilon}_{om}$, which by claim (b) is $\top$ or evaluates to `true` exactly for those objects.

Predicate comparisons. For $\text{eq}(p, p')$: B contains all p - and p' -triples of v , and compile_n retains poms_p and $\text{poms}_{p'}$ unfiltered; these generate all of them. For $\text{disj}(p, p')$, $\text{lessThan}(p, p')$, $\text{lessThanEq}(p, p')$, $\text{uniqueLang}(p)$, and $\text{closed}(P)$: $B = \emptyset$ (Table 5), so the claim is vacuous. For $\neg\text{eq}(p, p')$: B retains the p -triples whose object is not a p' -value and symmetrically, and compile_n injects $f_{\text{disj}}(om, om_{p'})$, which evaluates to `true` exactly for those objects (Definition 18). For $\neg\text{disj}(p, p')$, $\neg\text{lessThan}(p, p')$, and $\neg\text{lessThanEq}(p, p')$: B retains a p -triple (v, p, x) whenever some p' -value y satisfies $x = y$, $x \not\prec y$, respectively $x \not\preceq y$, and symmetrically for p' -triples (Table 5); compile_n injects $f_{\text{eq}}^\exists$, $f_{\text{geq}}^\exists$ and $f_{\text{leq}}^\exists$, respectively $f_{\text{gt}}^\exists$ and $f_{\text{lt}}^\exists$, whose conditions $\exists s: x = s, x \not\prec s, s \not\prec x, x \not\preceq s$, and $s \not\preceq x$ (Definition 18) coincide with those of B over the value sets, so the filter passes exactly for the retained objects. For $\neg\text{uniqueLang}(p)$: B retains each p -triple (v, p, x) for which some other p -value $y \neq x$ shares its language tag. Any such y is the value of another object map in poms_p declaring the same tag as the object map of x , so the pom of x is selected by Table 3 and the triple is generated. For $\neg\text{closed}(P)$: $B = \{(v, q, o) \in G \mid q \notin P\}$. As φ contains $\neg\text{closed}(P)$, $P(\varphi)$ contains all predicates not in P (Section 2.2), so pruning retains every pom whose predicate is outside P , compile_n selects exactly those and generates all triples of B .

Boolean combinations. For $\varphi_1 \wedge \varphi_2$: $B(v, G, \varphi_1 \wedge \varphi_2) = B(v, G, \varphi_1) \cup B(v, G, \varphi_2)$, and compile_n returns the union of both results; by induction hypothesis each part is retained. When a pom is produced both with and without a function on its object map, `dedup` keeps the version without, which subsumes the other (Algorithm 2). For $\varphi_1 \vee \varphi_2$: B is again the union, where $B(v, G, \varphi_i) = \emptyset$ whenever $v \not\models \varphi_i$. compile_n gates the contribution of each disjunct by $\varepsilon_i = \text{compile}_v(\varphi_i, map, tm, \mathcal{H})$. By claim (a) applied to φ_i , a satisfied disjunct yields $\varepsilon_i \neq \perp$ that does not evaluate to `false` on r_v , so its poms are not suppressed by resolve_n on the records generating v . For $\text{hasShape}(s')$ and $\neg\text{hasShape}(s')$: compile_n unfolds to $\text{exp}(s', \mathcal{H})$, respectively its negation, and the result follows by induction hypothesis.

Constants and node tests. For $\text{test}(t)$, $\text{hasValue}(c)$, and their negations: $B = \emptyset$, so the claim is vacuous.

Target declarations. For $\tau = \geq_1 \mathbf{a}.\text{hasValue}(c)$: $B(v, G, \tau) = \{(v, \mathbf{a}, c)\}$. The $\geq_n p.\varphi'$ entry of Table 3 applies to $\text{poms}_{\mathbf{a}}(tm)$ with $\varepsilon_{om} = \text{compile}_v(\text{hasValue}(c), om)$; for the retained pom $(\text{const}(\mathbf{a}), \text{const}(c))$ (Definition 11 case 1) this statically folds to $\top$, so the pom is retained unfiltered and generates $(v, \mathbf{a}, c)$ on r_v ; any further $\text{rdf:type } pom$, present only when $\text{rdf:type} \in P(\varphi)$, folds to $\perp$ and is removed from the τ contribution, its triples being covered, when in $B(v, G, \varphi)$, by the φ contribution and **dedup**. For $\tau = \geq_1 p_\tau.\top$: $B(v, G, \tau)$ consists of all p_τ -triples of v ; the same entry applies with $\varepsilon_{om} = \text{compile}_v(\top, om) = \top$, so poms_{p_τ} is retained unfiltered and, applied to p_τ (whose $poms$ are retained by Definition 11 case 2), generates all of them. For $\tau = \geq_1 p_\tau^-. \top$: $B(v, G, \tau)$ consists of all triples $(x, p_\tau, v) \in G$. The $\geq_n p^-.\top$ entry of Table 3 contributes, on each invocation with $map \in \text{fmaps}(tm, \tau)$, the pom of map itself, so accumulating over $\text{fmaps}(tm, \tau)$ (Algorithm 2) yields every p_τ - pom of tm , each carrying its validation injection. By Lemma 1 every triple $(x, p_\tau, v) \in G$ is generated by one of these $poms$ on some record, and by claim (a), instantiated at the value v , the injection passes there since $v \models \varphi$. Hence all witnesses are generated.

Thus every triple in $B(v, G, \varphi \wedge \tau)$ is in $\llbracket \Sigma'_{\text{EV}}(s, \varphi, \tau) \rrbracket$.

D.6 Proof of Lemma 6 (Neighborhood Compilation Soundness)

Proof. Let $(s, \varphi, \tau) \in \mathcal{H}$ and let v with $G, v \models \tau$ and $G, v \models \varphi$. By the same structural induction as Lemma 5, we verify that compile_n retains no triple of v outside $B(v, G, \varphi \wedge \tau)$. By Algorithm 2, the $poms$ of the rewritten triples map are exactly those returned by compile_n after **dedup**, while the subject map, and for $\tau = \geq_1 p_\tau^-.\top$ the object maps, retain the validation injection; so it suffices to consider the $poms$ each case of Table 3 contributes for φ and for τ .

Path quantifiers. For $\forall p.\varphi'$: compile_n contributes poms_p , matching B , which contains all p -triples of v since φ' is path free. For $\geq_n p.\varphi'$: for an object x not satisfying φ' , ε_{om} is $\perp$ or evaluates to **false** on r_v by claim (b), so resolve_n removes the pom or injects a filter returning **none**; such triples are not in B , and all retained ones are. For $\leq_n p.\varphi'$: symmetrically, objects satisfying φ' are suppressed by $\hat{\cdot} \varepsilon_{om}$, using claim (a), and are not in B .

Predicate comparisons. For $\text{eq}(p, p')$: compile_n contributes $\text{poms}_p \cup \text{poms}_{p'}$, matching B . For $\text{disj}(p, p')$, $\text{lessThan}(p, p')$, $\text{lessThanEq}(p, p')$, $\text{uniqueLang}(p)$, $\text{closed}(P)$, $\text{test}(t)$, $\text{hasValue}(c)$, and the negations of the latter two: no entry of Table 3 applies, so nothing is contributed, matching $B = \emptyset$. For $\neg \text{eq}(p, p')$, $\neg \text{disj}(p, p')$, $\neg \text{lessThan}(p, p')$, and $\neg \text{lessThanEq}(p, p')$: the injected f_{disj} , $f_{\text{eq}}^\exists$, $f_{\text{geq}}^\exists/f_{\text{leq}}^\exists$, and $f_{\text{gt}}^\exists/f_{\text{lt}}^\exists$ pass a generated object x exactly under the conditions $x \notin S$, $\exists s: x = s$, $x \not\prec s$, $s \not\prec x$, $x \not\preceq s$, and $s \not\preceq x$ (Definition 18), which over the value sets are exactly the retention conditions of Table 5; all other triples are suppressed by the f_{passif} wrapper. For $\neg \text{uniqueLang}(p)$: a selected pom shares its declared language tag with another object map of poms_p ; on r_v , whenever both yield values, these are distinct and share the tag, so each generated triple (v, p, x) has a witness

$y \neq x$ with $y \sim x$ and lies in B ; if the other object map yields no value, the generated triple is again covered only under the degenerate-case exclusion. For $\neg\text{closed}(P)$: the contributed *poms* carry predicates outside P , so every generated triple (v, q, o) has $q \notin P$ and lies in B .

Boolean combinations. For $\varphi_1 \wedge \varphi_2$: both contributions are sound by induction hypothesis, and their union matches $B(v, G, \varphi_1) \cup B(v, G, \varphi_2)$. **dedup** keeps a *pom* unfiltered only when some case contributes it unfiltered, whose triples already lie in B by the corresponding induction case. For $\varphi_1 \vee \varphi_2$: for a disjunct with $v \not\models \varphi_i$, claim (b) gives $\varepsilon_i = \perp$ or an evaluation of **false** on r_v , so **resolve_n** removes its *poms* or suppresses their triples of v , matching $B(v, G, \varphi_i) = \emptyset$; a satisfied disjunct is sound by induction hypothesis. For **hasShape**(s') and $\neg\text{hasShape}(s')$: by induction hypothesis on the unfolded expression.

Target declarations. For $\tau = \geq_1 \mathbf{a}.\text{hasValue}(c)$: the contribution of τ is the *pom* $(\text{const}(\mathbf{a}), \text{const}(c))$, whose triple $(v, \mathbf{a}, c)$ is in $B(v, G, \tau)$. For $\tau = \geq_1 p_\tau.\top$: the contribution is **poms_{p_τ}** unfiltered, whose triples of v are exactly the p_τ -triples of v , all in $B(v, G, \tau)$. For $\tau = \geq_1 p_\tau^-. \top$: the contributed *poms* are the p_τ -*poms* carrying the validation injection; a triple with object v is generated only where the injection passes, and every such triple (x, p_τ, v) lies in $B(v, G, \tau)$.

Thus every triple of v in $\llbracket \Sigma'_{\text{EV}}(s, \varphi, \tau) \rrbracket$ is in $B(v, G, \varphi \wedge \tau)$.

Appendix E Test Suite Statistics

The test suite comprises 50 test cases covering all supported SHACL constructs and their combinations, together with the consolidation steps they exercise and source data with null and mixed-conformance values. Each case is executed on CSV, JSON, XML, and relational sources under both $\mathcal{H}$ and $\neg\mathcal{H}$, yielding 400 executions. pySHACL-fragments [9] supports 35 of the 50 cases under $\mathcal{H}$ and 2 under $\neg\mathcal{H}$. The full statistics table of the 50 test cases, with the constructs and source characteristics of each, is available in the SHACL2RML repository¹. The GTFS schema $\mathcal{H}$ used for RQ2 comprises 143 constraints: 58 **sh:nodeKind**, 46 **sh:datatype**, 16 **sh:pattern**, 12 **sh:class**, 5 **sh:minInclusive**, 4 **sh:minCount**, 1 **sh:node**, and 1 **sh:maxCount**.