Runtime Enforcement Using Knowledge Bases

Eduard Kamburjan[1](✉) and Crystal Chang Din[2]

[1] University of Oslo, Oslo, Norway
eduard@ifi.uio.no
[2] University of Bergen, Bergen, Norway
crystal.din@uib.no

Abstract. Knowledge bases have been extensively used to represent and reason about *static* domain knowledge. In this work, we show how to enforce domain knowledge about *dynamic processes* to guide executions at runtime. To do so, we map the execution trace to a knowledge base and require that this mapped knowledge base is always consistent with the domain knowledge. This means that we treat the *consistency* with domain knowledge as an invariant of the execution trace. This way, the domain knowledge guides the execution by determining the next possible steps, i.e., by exploring which steps are possible and rejecting those resulting in an inconsistent knowledge base. Using this invariant directly at runtime can be computationally heavy, as it requires to check the consistency of a large logical theory. Thus, we provide a transformation that generates a system which is able to perform the check only on the *past* events up to now, by evaluating a smaller formula. This transformation is transparent to domain users, who can interact with the transformed system in terms of the domain knowledge, e.g., to query computation results. Furthermore, we discuss different mapping strategies.

1 Introduction

Knowledge bases (KBs) are logic-based representations of both data and domain knowledge, for which there exists a rich toolset to query data and reason about data *semantically*, i.e., in terms of the domain knowledge. This enables domain users to interact with modern IT systems [39] without being exposed to implementation details, as well as to make their domain knowledge available for software applications. KBs are the foundation of many modern innovation drivers and key technologies: Applications range from Digital Twin engineering [31], over industry standards in robotics [23] to expert systems, e.g., in medicine [38].

The success story of KBs, however, is so far based on the use of domain knowledge about *static* data. The connection to transition systems and programs beyond Prolog-style logic programming has just begun to be explored. This is mainly triggered by tool support for developing applications that *use* KBs [7,13,28], in a type-safe way [29,32].

In this work, we investigate how one can use domain knowledge about dynamic processes and formalize knowledge about the order of computations to be performed. More concretely, we describe a runtime enforcement technique to use domain knowledge to *guide* the selection of rules in a transition system, for

© The Author(s) 2023
L. Lambers and S. Uchitel (Eds.): FASE 2023, LNCS 13991, pp. 220–240, 2023.
https://doi.org/10.1007/978-3-031-30826-0_12

example to simulate behavior with respect to domain knowledge, a scenario that we use as a guiding example in this article, or to enforce compliance of business process models with respect to restrictions arising from the domain [41].

Approach. At the core, our approach considers the *execution trace* of a run, i.e., the sequence of rule applications, as a KB itself. As such, it can be combined with the KB that expresses the domain knowledge of *dynamic processes* (DKDP). The DKDP expresses knowledge about (partial) executions such that the execution trace must be consistent with it before and after every rule application. For example, in a simulation system for geology, the DKDP may express that a certain rock layer A is above a certain rock layer B and, thus, the event to deposit a layer must occur for B, before it occurs for A. Consistency with the DKDP forms a *domain invariant* for the trace of a system, i.e., a *trace property.*

To trigger a transition rule, we use a hypothetical execution step: the execution trace is extended with a potential event and the consistency of the extended trace against the DKDP is checked. However using this consistency invariant directly at run time can be computationally heavy, as it requires to check the consistency of a large logical theory. Thus, we give a transformation that removes the need for a hypothetical execution step and instead results in a transition system that evaluates a transformed condition on (1) the *existing* trace and (2) the parameters of the potentially extended event. This condition does not require domain-specific reasoning anymore. This transformation removes the need for hypothetical execution steps and DKDP can be used to guide *any* transition system, including languages based on structural operational semantics. For example, it is then possible to express the invariant checking as a guard for the rule that deposits layers (e.g., only deposit A if layer B has been deposited already).

It is crucial that this system is usable for both the domain user (who possesses the domain knowledge) and the programmer (that has to program the interaction with the domain knowledge), a requirement explicitly stressed by Corea et al. [16] for the use of ontologies in business process models. We, thus, carefully designed our framework to increase its usability: First, the reasoning (in the geology example above, from spatial properties of layers to temporal properties of events) is completely performed in the domain and needs not be handled by the transition system. I.e., the programmer must not perform reasoning over the KB in the program itself. Second, the DKDP is expressed over *domain events*, as the domain users do not have knowledge about implementation details, such as the state organization. Furthermore, the formalization of the DKDP should not be affected by the underlying implementation details such that the DKDP can be reused. The DKDP can reuse the aforementioned industry standards and established ontologies, as well as modeling languages and techniques from ontology engineering [17], such as OWL [42], which are established for domain modeling and more suitable for this task than correctness-focused temporal logics such as LTL [35]: The domain users must not be an expert in programming or verification to contribute to the system.

The transformation that replaces the need for a hypothetical execution step with a transition system evaluating a transformed condition is also transparent

to the domain users. We say a transformed guarded rule is applicable if it would not violate consistency w.r.t. the DKDP. Lastly, we provide the domain users possibilities to query the final result, i.e., the KB of the final execution trace, and to explore possible simulations using the defined DKDP. Note that the mapping from trace to KB must not necessarily be designed manually: various (semi-) automatic mapping design strategies are discussed in the paper.

Contributions and Structure. Our main contributions are (1) a system that enforces domain knowledge to guide a transition system at runtime, and (2) a procedure that transforms such a transition system that uses consistency with domain knowledge as an invariant into a transition system using first-order guards over past events in a transparent way. We give preliminaries in Sec. 2 and present our running example in Sec. 3. We formalize our approach in Sec. 4 and give the transformation in Sec. 5, before we discuss (semi-)automatically generated mappings in Sec. 6. We discuss the mappings in Sec. 7 and related work in Sec. 8. Lastly, Sec. 9 concludes.

2 Preliminaries

We give some technical preliminaries for knowledge bases as well as transition systems, as far as they are needed for our runtime enforcement technique.

Definition 1 (Domain Knowledge of Dynamic Processes). Domain knowledge of dynamic processes *(DKDP) is the knowledge about events and changes.*

Example 1 (DKDP in Geology). DKDP describes knowledge about some temporal properties in a domain. In geology, for example, this may be the knowledge that a deposition of some geological layers in Cretaceous should happen after a deposition in Jurassic, because the Cretaceous is after the Jurassic. This can be deduced from, e.g., fossils found in the layers.

A description logic (DL) is a decidable fragment of first-order logic with suitable expressive power for knowledge representation [3]. We do not commit to any specific DL here, but require that for the chosen DL it is decidable to check consistency of a KB, which we define next. A knowledge base is a collection of DL axioms, over individuals (corresponding to first-order logic constants), *concepts*, also called classes (corresponding to first-order logic unary predicates) and *roles*, also called properties (corresponding to first-order logic binary predicates).

Definition 2 (Knowledge Base). *A knowledge base (KB)* $\mathcal{K} = (\mathcal{R}, \mathcal{T}, \mathcal{A})$ *is a triple of three sets of DL axioms, where the* ABox $\mathcal{A}$ *contains assertions over individuals, the* TBox $\mathcal{T}$ *contains axioms over concepts, and the* RBox $\mathcal{R}$ *contains axioms over roles. A KB is consistent if no contradiction follows from it.*

KBs can be seen as first-order logic theories, so we refrain from introducing them fully formally and introduce them by examples throughout the article. The Manchester syntax [25] is used for DL formulas in examples to emphasize that they model knowledge, but we treat them as first-order logic formulas otherwise.

Example 2. Continuing Exp. 1, the following axiom, expressing that Jurassic is before Cretaceous, is expressed by the following ABox axiom, where `Jurassic` and `Cretaceous` are individuals, while `before` is a role.

$$\texttt{before}(\texttt{Jurassic}, \texttt{Cretaceous})$$

The following TBox axioms express that every layer with Stegosaurus fossils has been deposited during the Jurassic. The first two axioms define the concepts `StegoLayer` (the class of things having the value `Stegosaurus` as their `contains` role) and `JurassicLayer` (the class of things having the value `Jurassic` as their `during` role). The last axiom says that the class of things having the value `Stegosaurus` as their `contains` role is a subclass of `JurassicLayer`. [3] The bold literals are keywords, the literals `StegoLayer`, `JurassicLayer` denote concepts/classes, the literals `contains`, `during` denote roles/properties and the literals `Stegosaurus`, `Jurassic` denote individuals.

```
StegoLayer EquivalentTo contains value Stegosaurus
JurassicLayer EquivalentTo during value Jurassic
StegoLayer SubClassOf JurassicLayer
```

The following RBox axioms express two constraints: The first line states that both `below` and `before` roles are asymmetric. The second line states that if a deposition is from an age before the age of another deposition, then it is below that deposition. Formally, the axiom expresses that the concatenation of the following three roles (a) the `during` role, (b) the `before` role, and (c) the inverse of the `during` role, is the sub-property of the `below` role. I.e., given an individual a, every individual b reachable from a following the chain `during`, `before` and the inverse of `during`, is also reachable by just `below`.

```
Asy(below)    Asy(before)
 during o before o inverse(during) SubPropertyOf below
```

Knowledge based guiding can be applied to any transition system to leverage domain knowledge during execution. States are not the focus of our work, and neither is the exact form of the rules that specify the transition between states. For our purposes, it suffices to define states as terms, i.e., finite trees where each node is labeled with a name from a finite set of term symbols, and transition rules as transformations between schematic terms. State guards can be added but are omitted for brevity's sake.

Definition 3 (Terms and Substitutions). *Let Σ_T be a finite set of term labels and Σ_V a disjoint set of term variables. A* term *t is a finite tree, where each inner node is a term label and each leaf is either a term label or a term variable. The set of term variables in a term t is denoted $\Sigma(t)$. We denote the set of all terms with T. A* substitution *σ is a map from term variables to terms without term variables. The application of a substitution σ to a term t, with the usual semantics, is denoted $t\sigma$. In particular, if t contains no term variables, then $t\sigma = t$.*

[3] The first-order equivalent is $\forall x.\ \texttt{contains}(x, \texttt{Stegosaurus}) \rightarrow \texttt{during}(x, \texttt{Jurassic})$

Rewrite rules map one term to another by unifying a subterm with the head term. The matched subterm is then rewritten by applying the substitution to the body term. Normally one would have additional conditions on the transition rules, but these are not necessary to present semantical guiding.

Definition 4 (Term Rewriting Systems). *A* transition rule *in the term rewriting system has the form*

$$t_{\texttt{head}} \xrightarrow{\mathsf{r}} t_{\texttt{body}}$$

Where r *is the name of the rule, and* $t_{\texttt{head}}, t_{\texttt{body}} \in T$ *are the head and body terms.*

A rule matches *on a term* t *with* $\Sigma(t) = \emptyset$*, if there is a subterm* t_s *of* t*, such that* $t_{\texttt{head}} = t_s\sigma$*, for a suitable substitution* σ*. A rule* produces *a term* t'*, by matching on subterm* t_s *with substitution* σ*, and generating* t' *by replacing* t_s *in* t *by* $t_s\sigma'$*, where* σ' *is equal to* σ *for all* $v \in \Sigma(t_{\texttt{body}}) \cap \Sigma(t_{\texttt{head}})$ *and maps* $v \in \Sigma(t_{\texttt{head}}) \setminus \Sigma(t_{\texttt{body}})$ *to fresh term symbols. For production, we write*

$$t \xrightarrow{\mathsf{r},\sigma'} t'$$

3 A Scenario for Knowledge Based Guiding

To illustrate our approach, we continue with geology, namely with a simulator for deposition and erosion of geological layers. Such a simulator is used, e.g., for hydrocarbon exploration [20]. It contains domain knowledge about the type of fossils and the corresponding geological age, and connects spatial information about deposition layers with temporal information about their deposition. We started a formalization of the DKDP in Ex. 2 and expand it below.

The core challenge is that the simulator must make sure that it does not violate domain properties. This means that it cannot deposit a layer containing fossils from the Jurassic after depositing a layer containing fossils from the Cretaceous. This information is given by the domain users as an *invariant*, i.e., as knowledge that the execution must be consistent with at all times.

Programming with Knowledge Bases. Our model of computation is a set of rewrite rules on some transition structure. The sequence of rule applications, denoted *events*, forms the trace. DKDP constrains the extension of the trace. This realizes a clear separation of concerns between *declarative* data modelling and *imperative* programming with, in our case, transitions.

Example 3. Let us assume 4 rules: a rule `deposit` that deposits a layer without fossils, a rule `depositStego` that deposits a layer with Stegosaurus fossils, an analogous rule `depositTRex` that deposits a layer with Tyrannosaurus fossils, and a rule `erode` that removes the top layer of the deposition. One example reduction sequence, for some terms t_i and with substitutions omitted, is as follows:

$$t_0 \xrightarrow{\texttt{depositStego}} t_1 \xrightarrow{\texttt{erode}} t_2 \xrightarrow{\texttt{depositTRex}} t_3$$

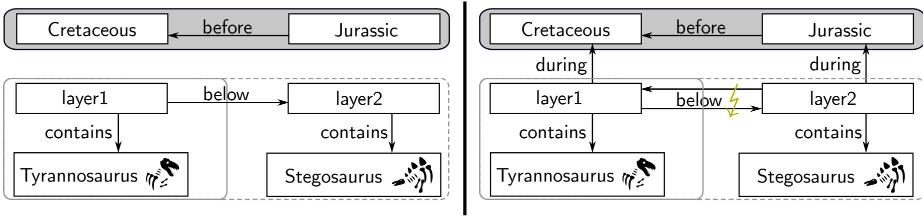

Fig. 1. Left: KB as generated. Right: Inferred KB to detect inconsistency.

which describes the rule application of `depositStego` on term t_0 following by the rule application of `erode` on term t_1 and then `depositTRex` on term t_2.

In the domain KB, we add an axiom expressing that the geological layer containing Stegosaurus fossils is deposited during the Jurassic, and that the geological layer containing Tyrannosaurus fossils is deposited during the Cretaceous.

Consider that rule `depositStego` may trigger on term t_3.

$$\dots t_2 \xrightarrow{\texttt{depositTRex}} t_3 \xrightarrow[?]{\texttt{depositStego}}$$

This would violate the domain knowledge, as we can derive a situation, where a layer with Tyrannosaurus fossils is below a layer with Stegosaurus fossils, implying that the Cretaceous is before the Jurassic. This contradiction is captured by the knowledge base in Fig. 1. The domain knowledge DKDP should prevent this rule application at t_3 to happen. To achieve this, i.e., enforce domain knowledge at runtime, we must connect the trace with the KB. Specifically, we represent the trace as a KB itself, i.e., instead of operating on a KB, we record the events and generate a KB from a trace using a mapping.

For example, consider the left KB in Fig. 1. The upper part is (a part of) our DKDP about geological ages, while the lower part is the KB mapped from the trace. Together they form a KB. In the knowledge base of this example, we add one layer that contains Stegosaurus fossils for each `depositStego` event and analogously for `depositTRex` events. We also add the `below` relation between two layers, if their events are ordered. So, if we would execute `depositStego` after `depositTRex`, there would be two layers in the KB as shown in Fig. 1, with corresponding fossils, connected using the `below` relation. On the right, the KB is shown with the additional knowledge following from its axioms. In particular, we can deduce that layer2 must be below layer1 using the axioms from Sec. 2. This, in turn, makes the overall KB inconsistent, as `below` must be asymmetric.

We stress that consistency of the execution with the DKDP is a trace property, it is reasoning about the events that happen regardless of the current state. In our example, consider the situation, where the next event after t_3 rule `erode` triggers again, and then we consider rule `depositStego`. I.e., the following continuation of the trace

$$\dots t_2 \xrightarrow{\texttt{depositTRex}} t_3 \xrightarrow{\texttt{erode}} t_4 \xrightarrow[?]{\texttt{depositStego}}$$

We still consider the layer with the Tyrannosaurus fossils in our KB, despite its erosion. Firstly, because the layer may potentially have had an effect on the execution before being removed, and, secondly, because its deposition also models implicit information. It expresses the current geological era of the system, which cannot be reverted: at t_3 the system is in the Cretaceous, and while the `depositStego` models an action in the Jurassic – the trace would not represent a semantically sensible execution if the `depositStego` rule would be executed.

Fig. 2 illustrates the runtime enforcement of domain knowledge on traces in a more general setting. The execution itself is a reduction sequence over some terms t, where each rule application emits some event `ev`, e.g., name of the applied rule and matched subterms. A *mapping* μ is used to generate a KB from the trace. The knowledge base then contains (a) the DKDP, pictured as the shaded box, (b) the mapping of the trace so far, pictured as the unshaded box with solid frame, and (c) the potential next event, pictured as the dashed box. Additionally, new connections may be inferred.

The mapping from a trace to a KB matches the system formalized by the domain knowledge to the system used for *programming*, it is the interface between domain experts and the programmer. Indeed, the mapping allows the domain users to investigate program executions without being exposed to the implementation details. Given a fixed execution, the mapping can be applied to allow the domain users to query its results (in form of the trace) using domain vocabulary.

From the program's point of view, it defines an invariant over the trace, which must always hold: consistency with domain knowledge. While this saves the domain users from learning about the implementation, it poses two challenges to the programmer: first, the mapping must be developed additionally to the rules, and second, the invariant is not specific to the rules. The *extended* trace caused by the execution of one single event, must be checked against the full DKDP, which is not specific to any transition event. Instead of this computationally costly operation, we provide an alternative. For example, to ensure consistency when executing the rule `depositStego`, it suffices to evaluate the following formula on the *past* trace *tr* to check that `depositTRex` has not been executed yet: $\forall i \leq |tr|.\ tr[i] \neq \texttt{ev}(\texttt{depositTRex})$. The condition of a rule is specific to the corresponding transition action, instead of a general condition on all the rules.

After defining runtime enforcement of domain knowledge formally, we will return to these challenges and (a) discuss different mapping strategies, and especially the (semi-)automatic generation of mappings and (b) give a system that, for a big class of mappings, also derives local conditions.

4 Knowledge Guided Transition Systems

We now introduce runtime enforcement using KBs. To this end, we define the mapping of traces to KBs formally and give the transition system that uses this lifting for consistency checking. First, we define the notion of traces.

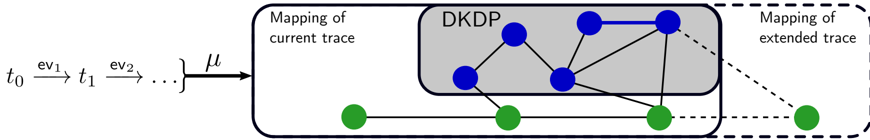

Fig. 2. Runtime enforcement of knowledge bases on traces.

Definition 5 (Execution Traces). *An* event *ev for a rule r and a substitution σ has the form* $\mathsf{ev}(r, \sigma)$*, which we write as* $\mathsf{ev}(r, v_1 : t_1, \ldots, v_n : t_n)$*, where $v_i : t_i$ are the pairs in σ. To record the sequence of an execution, we use* traces*. A trace is a finite sequence of events, where each event records the applied rule and the corresponding substitutions, if there are any.*

Example 4. The trace of the rule application in Ex. 3 is as follows, for suitable substitutions that all store the deposited or eroded layer in the variable v.

$$\left\langle \mathsf{ev}\big(\mathtt{depositStego}, v : \mathtt{layer0}\big), \mathsf{ev}\big(\mathtt{erode}, v : \mathtt{layer0}\big), \mathsf{ev}\big(\mathtt{depositTRex}, v : \mathtt{layer1}\big)\right\rangle$$

To connect executions with knowledge bases, we define mappings that transform traces into knowledge bases, given a fixed vocabulary Σ.

Definition 6 (Mappings). *A Σ-*mapping *μ is a function from traces to knowledge bases over vocabulary Σ.*

The mapping is given by the user, who has to respect the signature of the KB formalizing the used domain knowledge. While we are not specific in the structure of the mapping in general, we introduce the notion of a *first-order matching mapping*, which allow for optimization and automatization.

Definition 7 (First-Order Matching Mapping). *A* first-order matching mapping *μ is defined by a set $\{\varphi_1 \mapsto_{N_1} ax_1, \ldots, \varphi_n \mapsto_{N_n} ax_n\}$, where each element has a first-order logic formula φ_i as its guard, a set of individuals N_i and some set ax_i of KB axioms as its body. We write $ax_i(N)$ to emphasize that a set of individuals N occur in $ax_i(N)$.*

The mapping is applied to a trace tr by adding all those bodies whose guard evaluates to true and replacing all members of N in ax_1 by fresh individual names:

$$\mu(tr) = \left(\bigcup_{tr \models \varphi_i} ax_i(N) \right) [N \textit{ fresh}]$$

Where $A[N \textit{ fresh}]$ substitutes all individuals in N with fresh names in A.

Example 5. Consider the following first-order matching mapping μ, for some role/property $\mathtt{P}$ and individuals $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{C}$. The function $\mathsf{rule}(ev)$ extracts the rule name from the given event ev.

$$\big\{\exists i.\ \mathsf{rule}(tr[i]) \doteq \mathsf{r}_1 \mapsto_\emptyset \mathtt{P}(\mathtt{A}, \mathtt{B}), \quad \exists i.\ \mathsf{rule}(tr[i]) \doteq \mathsf{r}_2 \mapsto_\emptyset \mathtt{P}(\mathtt{B}, \mathtt{A}),$$
$$\exists i.\ \mathsf{rule}(tr[i]) \doteq \mathsf{r}_3 \mapsto_\emptyset \mathtt{P}(\mathtt{A}, \mathtt{C}), \quad \exists i.\ \mathsf{rule}(tr[i]) \doteq \mathsf{r}_4 \mapsto_\emptyset \mathtt{P}(\mathtt{C}, \mathtt{A})\big\}$$

Its application to a trace $\langle \mathsf{ev}(\mathsf{r}_1), \mathsf{ev}(\mathsf{r}_1), \mathsf{ev}(\mathsf{r}_2)\rangle$ is the set $\{\mathtt{P}(\mathtt{A},\mathtt{B}), \mathtt{P}(\mathtt{B},\mathtt{A})\}$.

First-order matching mapping can also be applied to our running example.

Example 6. We continue with the trace from Ex. 4, extended with another event $\mathsf{ev}(\mathtt{depositStego}, v : \mathtt{layer2})$. We check whether adding an event to the trace would result in a consistent KB by actually extending the trace for analysis. We call this a hypothetical execution step.

The following mapping, which must be provided by the user adds the spatial information about layers w.r.t. the fossils found within. The first-order logic formula at the guard of the mapping expresses that an event of `depositTRex` is found before the event of `depositStego` in the trace. Note that the given set of axioms from the mapping faithfully describes the *event structure of the trace*, i.e., the mapping could produce axioms which will cause inconsistency w.r.t. the domain knowledge: Together with the DKDP, we can see that the trace is mapped to an inconsistent knowledge base by adding 5 axioms. Note that we do not generate one layer for each deposition event during simulation, but only two specific ones, $\mathtt{Layer}(\mathtt{l}_1)$ and $\mathtt{Layer}(\mathtt{l}_2)$ in this case, for the relevant information. One can extend mapping rules for the different cases (for instance, `depositStego` before `depositTRex`, only `depositTRex` events, etc.), or use a different mapping mechanism, which we discuss further in Sec. 6.

$$
\begin{aligned}
&\exists l_1, l_2.\ \exists i_1, i_2.\\
&\quad tr[i_1] \doteq \mathsf{ev}\big(\mathtt{depositTRex}, v : l_1\big) \wedge tr[i_2] \doteq \mathsf{ev}\big(\mathtt{depositStego}, v : l_2\big) \wedge i_1 < i_2\\
&\quad \mapsto_{\mathtt{l}_1,\mathtt{l}_2}\\
&\big\{\mathtt{Layer}(\mathtt{l}_1), \mathtt{contains}(\mathtt{l}_1, \mathtt{Tyrannosaurus}),\\
&\ \ \mathtt{Layer}(\mathtt{l}_2), \mathtt{contains}(\mathtt{l}_2, \mathtt{Stegosaurus}), \mathtt{below}(\mathtt{l}_1, \mathtt{l}_2)\big\}
\end{aligned}
$$

We stress again that we are interested in trace properties, a layer may still have had effects on the state despite being completely removed at one point (by an `erode` event). Thus, we must consider the *deposition event* of a layer to check the trace against the domain knowledge.

The guided transition systems extends the mapping of a basic transition system, by additionally ensuring that the trace *after* executing the rule would be mapped to a consistent knowledge base. This treats the domain knowledge as an invariant that is enforced, i.e., a transition is only allowed if it indeed preserves the invariant.

Definition 8 (Guided Transition System). *Given a set of rules $\mathcal{R}$, a mapping μ and a knowledge base $\mathcal{K}$, the guided semantics is defined as a transition system between pairs of terms t and traces tr. For each rule $\mathsf{r} \in \mathcal{R}$, we have one guided rule (for consistency, cf. Def. 2):*

$$
(\textsf{kb})\ \frac{t \xrightarrow{\mathsf{r},\sigma} t' \qquad ev = \mathsf{ev}(\mathsf{r},\sigma) \qquad \mu(tr \circ ev) \cup \mathcal{K} \textit{ is consistent}}{(t, tr) \xrightarrow{\mathsf{r}} (t', tr \circ ev)}
$$

The set of traces generated by a rewrite system $\mathcal{R}$ from a starting term t_0 is denoted $\mathbf{H}(\mathcal{R}, \mu, \mathcal{K}, t_0)$. Execution always starts with the empty trace.

5 Well-Formedness and Optimization

The transition rule in Def. 8 uses the knowledge base directly to check consistency, and while this enables to integrate domain knowledge into the system directly, it also poses challenges from a practical point of view. First, the condition of the rule application is not specific to the change of the trace, and must check the consistency of the whole knowledge base, which can be computationally heavy. Second, the consistency check is performed at every step, for every potential rule application. Third, the trace must be mapped whenever it is extended. Which means the same mapping computation that has been performed in the previous step may be executed all over again.

To overcome these challenges, we provide a system that reduces consistency checking by using *well-formedness* guards, which only require to evaluate an expression over the trace without accessing the knowledge base. These guards are transparent to the domain users, the system behaves the same as with the consistency checks of the knowledge base. At its core, we use well-formedness predicates, which characterize the relation of domain knowledge and mappings.

Definition 9 (Well-Formedness). *A first-order predicate wf of a trace tr is a well-formedness predicate for some mapping μ and some knowledge base $\mathcal{K}$, if the following holds:*

$$\forall tr.\ wf(tr) \iff \mu(tr) \cup \mathcal{K} \textit{ is consistent}$$

Using this definition we can slightly rewrite the rule of Def. 8: For every starting term t_0, the set of generated traces is the same if the rule of Def. 8 is replaced by the following one

$$\textsf{(wf)}\ \frac{t \xrightarrow{\mathsf{r},\sigma} t' \qquad ev = \mathsf{ev}(\mathsf{r}, \sigma) \qquad wf(tr \circ ev)}{(t, tr) \xrightarrow{\mathsf{r}} (t', tr \circ ev)}$$

For first-order matching mappings, we can generate the well-formedness predicate by testing all possible extensions of the knowledge base upfront and defining the guards of those sets that are causing inconsistency as non-well-formed.

Theorem 1. *Let μ be a first-order matching mapping for some knowledge base $\mathcal{K}$. Let $\mathtt{Ax} = \{ax_1, \dots, ax_n\}$ be the set of all bodies in μ. Let $\mathtt{Incons}$ be the set of all subsets of $\mathtt{Ax}$, such that for each $A \in \mathtt{Incons}$, $\bigcup_{a \in A} a \cup \mathcal{K}$ is inconsistent. Let $\mathtt{guard}_A$ be the set of guards corresponding to each body in A. The following predicate wf_μ is a well-formedness predicate for μ and $\mathcal{K}$.*

$$wf_\mu = \neg \bigvee_{A \in \mathtt{Incons}} \bigwedge_{\varphi \in \mathtt{guard}_A} \varphi$$

Example 7. We continue with Ex. 5. Consider a knowledge base $\mathcal{K}$ expressing that role P is asymmetric. The knowledge base becomes inconsistent if the first two or the last two axioms from μ are added to the knowledge base. Thus, the generated well-formedness predicate *wf* is the following

$$wf_{\mu}(tr) \equiv \neg\Big(\big((\exists i.\ \mathsf{rule}(tr[i]) \doteq \mathsf{r}_1) \wedge (\exists i.\ \mathsf{rule}(tr[i]) \doteq \mathsf{r}_2)\big)\vee \\ \big((\exists i.\ \mathsf{rule}(tr[i]) \doteq \mathsf{r}_3) \wedge (\exists i.\ \mathsf{rule}(tr[i]) \doteq \mathsf{r}_4)\big)\Big)$$

The above procedure has exponential complexity in the number of branches of the mapping. But as the superset of an inconsistent set is also inconsistent, it is not necessary to generate all the subsets. I.e., it suffices to consider the following set of minimal inconsistencies instead, which can be computed by testing for inconsistencies based on the sets ordered by $\subset$.

$$\texttt{min-Incons} = \{A \mid A \in \texttt{Incons} \wedge \forall A' \in \texttt{Incons}.\ A' \neq A \rightarrow A' \not\subset A\}$$

If well-formedness is defined inductively, then we can give an even more specific transformation. The well-formedness predicate is inductive, if it checks well-formedness for each trace and its last event is equivalent to the evalution of a formula over the trace, which is *specific* to the event. If this is the case, then each rule, which dictates the event, can have an own, highly specialized well-formedness guard, which further enhances efficiency.

Definition 10 (Inductive Well-Formedness). *A well-formedness predicate wf is* inductive [4] *for some set of rules $\mathcal{R}$ if there is a set of predicates wf_r for all rules $r \in \mathcal{R}$, such that wf can be written as an inductive definition:*

$$wf(\langle\rangle) \equiv \mathsf{true}$$
$$wf(tr \circ ev) \equiv wf(tr) \wedge \bigwedge_{r \in \mathcal{R}} \big((\mathsf{rule}(ev) \doteq r) \rightarrow wf_r(tr, ev)\big)$$

in which $wf_r(tr, ev)$ is the local well-formedness predicate specifically for rule r with the condition $\mathsf{rule}(ev) \doteq r$. The predicate wf_r forms the guard for rule r. Every well-formedness predicate is equivalent to an inductive well-formedness predicate by setting $wf_r(tr, ev) = wf(tr \circ ev)$, but we aim to give more specific predicates per rule.

Example 8. Finishing our geological system, we can give local well-formedness predicates for all rules. For example, we can define a specific guard for rule `depositStego` expressing that the deposition of a layer containing Stegosaurus fossil is not allowed if there is already a deposition of a layer containing Tyrannosaurus fossils captured in the trace *tr* up to now. Compare with the approach that the whole knowledge base needs to be checked, this proposed solution using

[4] Our well-formedness predicates are inspired by the ones used in verification of concurrent systems, where they characterize traces w.r.t. a specific concurrency model [21].

inductive well-formedness simplifies the complexity of analysis significantly. For instance, the rule for deposition does not need to concern with the ordering of the geological age.

$$\begin{aligned} wf_{\texttt{deposit}}\big(tr, \mathsf{ev}(\texttt{deposit}, v:l)\big) &\equiv wf_{\texttt{erode}}\big(tr, \mathsf{ev}(\texttt{erode}, v:l)\big) \equiv \mathsf{true} \\ wf_{\texttt{depositTRex}}\big(tr, \mathsf{ev}(\texttt{depositTRex}, v:l)\big) &\equiv \mathsf{true} \\ wf_{\texttt{depositStego}}\big(tr, \mathsf{ev}(\texttt{depositStego}, v:l)\big) &\equiv \forall i \leq |tr|.\ \mathsf{rule}(tr[i]) \neq \texttt{depositTRex} \end{aligned}$$

Definition 11 (Transition System using Well-Formedness). *Let wf be an inductive well-formedness predicate for a set of rules $\mathcal{R}$, some mapping μ, some knowledge base $\mathcal{K}$. We define the transformed guarded transition system with the following rule for each $r \in \mathcal{R}$.*

$$(\textsf{wf-r})\ \frac{t \xrightarrow{\mathsf{r},\sigma} t' \qquad ev = \mathsf{ev}(\mathsf{r},\sigma) \qquad wf_{\mathsf{r}}(tr, ev)}{(t, tr) \xrightarrow{\mathsf{r}} (t', tr \circ ev)}$$

The set of traces generated by this transition system from a starting term t_0 is denoted $\mathbf{G}(\mathcal{R}, wf, t_0)$. Execution always starts with the empty trace.

Note that (a) we do use a specific well-formedness predicate per rule, and that (b) we do *not* extend the trace tr in the premise as the rules in Def. 8 and Def. 9.

Theorem 2. *Let wf be an inductive well-formedness predicate for a set of rules $\mathcal{R}$, some mapping μ, some knowledge base $\mathcal{K}$. The guided system of Def. 8 and Def. 11 generate the same traces: $\forall t.\ \mathbf{H}\big(\mathcal{R}, \mu, \mathcal{K}, t\big) = \mathbf{G}\big(\mathcal{R}, wf, t\big)$*

We can also define determinism as terms of the inductive well-formedness. An inductive well-formedness predicate wf is deterministic, if for each trace tr and event ev, only one possible local well-formedness predicate $wf_{\mathsf{r}}(tr, ev)$ holds.

Proposition 1 (Deterministic Well-Formedness). *An inductive well-formedness predicate wf with local well-formedness predicates $\{wf_{\mathsf{r}}\}_{r\in\mathcal{R}}$ is* deterministic*, if*

$$\forall tr.\ \forall ev.\ \bigwedge_{r\in\mathcal{R}} \Big(wf_r(tr, ev) \rightarrow \bigwedge_{\substack{r'\in\mathcal{R} \\ r'\neq r}} \neg wf_{r'}(tr, ev) \Big)$$

For deterministic predicates, only one trace is generated: $\big|\mathbf{G}\big(\mathcal{R}, wf, t\big)\big| = 1$.

When the programmer designs the mapping, the focus is on mapping enough information to achieve *inconsistency*, to ensure that certain transition steps are not performed. If the same mapping is to be used to retrieve results from the computation, e.g., to query over the final trace, this may be insufficient. Next, we discuss mappings that preserve more, or all information from the trace.

6 (Semi-)Automatically Generated Mappings

The mappings we discussed so far require to be defined completely by the programmer and are used to extract a certain correct information from a trace, which is sufficient to enforce domain invariants at runtime. In this section, we introduce mappings which can be constructed (semi-)automatically to simplify the usage of domain invariants: The *transducing mappings* and *direct mappings* leverage the structure of the trace directly. A *transducing mapping* is constructed semi-automatically. It applies some manually defined mapping to each event and automatically connects every pair of consecutive events in a trace using the `next` role in KB. A *direct mapping* relates each event with its parameters and is constructed fully automatically. Both kinds of mappings are not only easier to use for the programmer, they can also be used by the domain users to access the results of the computation in terms of the domain.

A transducing mapping is semi-automatic in the sense that part of the mapping is pre-defined, and the programmer must only define a part of it, namely the mapping from a single event to a KB.

Formally, a transducing mapping consists of a function ι that generates unique individual names[5] per event and a user-defined function ϵ that maps every event to a KB.

Definition 12 (Transducing Mapping). *Let ι an injective function from natural numbers to individuals, and ϵ be a function from events to KBs. Let* `next` *be an asymmetric role. Given a trace tr, a transducing mapping $\delta^{\texttt{next}}_{\iota,\epsilon}(tr)$ is defined as follows. For simplicity, we annotate the index i of an event in tr directly.*

$$\delta^{\texttt{next}}_{\iota,\epsilon}(\langle\rangle) = \emptyset \qquad \delta^{\texttt{next}}_{\iota,\epsilon}(\langle \mathsf{ev}_i \rangle) = \epsilon(\mathsf{ev}_i)$$

$$\delta^{\texttt{next}}_{\iota,\epsilon}(\langle \mathsf{ev}_i, \mathsf{ev}_j \rangle \circ tr) = \epsilon(\mathsf{ev}_i) \cup \{\texttt{next}(\iota(i), \iota(j))\} \cup \delta^{\texttt{next}}_{\iota,\epsilon}(\langle \mathsf{ev}_j \rangle \circ tr)$$

in which the $\circ$ operator concatenates two traces. This approach is less demanding than to design an arbitrary mapping, as the structure of the sequence between each pair of consecutive events is taken care of by the `next` role and ι is trivial in most cases: one can just generate a fresh node with the number as part of its individual symbol. The programmer only has to provide a function ϵ for events.

Example 9. Our geology example can be reformulated with the following user-defined function ϵ_{geo}. Let ι_{geo} map every natural number i to the symbol $\texttt{layer}_i$:

$$\epsilon_{geo}(\mathsf{ev}_i(\mathsf{depositStego}, v\!:\!l)) = \{\texttt{contains}(\iota_{geo}(i), \texttt{Stegosaurus}), \texttt{Layer}(\iota_{geo}(i))\}$$
$$\epsilon_{geo}(\mathsf{ev}_i(\mathsf{depositTRex}, v\!:\!l)) = \{\texttt{contains}(\iota_{geo}(i), \texttt{Tyrannosaurus}), \texttt{Layer}(\iota_{geo}(i))\}$$
$$\epsilon_{geo}(\mathsf{ev}_i(\mathsf{deposit}, v\!:\!l)) = \{\texttt{contains}(\iota_{geo}(i), \texttt{Nothing}), \texttt{Layer}(\iota_{geo}(i))\}$$
$$\epsilon_{geo}(\mathsf{ev}_i(\mathsf{erode})) = \emptyset$$

Note that the function $\iota_{geo}(i)$ is used to generate new symbols for each event, which are then declared to be geological layers by the axiom $\texttt{Layer}(\iota_{geo}(i))$. It

[5] If using the Resource Description Framework (RDF) [43] for the knowledge base, one requires fresh unique resource identifiers (URI).

generalizes the set of fresh names from first-order matching mappings in Def. 7. Based on this function definition, the example in Sec. 3 can be performed using the transducing mapping $\delta^{\texttt{below}}_{\iota_{geo},\epsilon_{geo}}$. The connections between each pair of consecutive events in a trace, i.e., a layer is `below` another layer, is derived from the axioms in the domain knowledge and is added as additional axioms to the KB.

So far, the mappings of the trace to some information in terms of a specific domain are defined by the programmer. To further enhance the automation of the mapping construction, we give a *direct* mapping, that captures *all* information of a trace in a KB. More technically, the *direct* mapping directly expresses the trace structure using a special vocabulary, which captures domain knowledge about traces *themselves* and is independent from any application domain. We first define the domain knowledge about trace structure.

Definition 13 (Knowledge Base for Traces). *The knowledge base for traces contains the concept* `Event` *modeling events, the concept* `Match` *modeling one pair of variable and its matching terms, and the concept* `Term` *for terms. Furthermore, the functional property* `appliesRule` *connects events to rule names (as strings), the property* `match` *that connects the individuals for events with the individuals for matches (i.e., an event with the pairs $v : t$ of a variable and the term assigned to this variable), the property* `var` *that connects matches and variables (as strings), and* `term` *that connects matches and terms.*

We remind that KBs only support binary predicates and we cannot avoid formalizing the concept of a match, which connects three parts: event, variable and term. The direct mapping lessens the workload for the programmer further: it requires no additional input and can be done fully automatically. It is a pre-defined mapping for *all programs* and is defined by instantiating a transducing mapping using the `next` role and pre-defined functions ϵ_{direct} and ι_{direct} for ϵ and ι. Also, we must generate additional fresh individuals for the matches. The formal definition of the pre-defined functions for the direct mapping is as follows.

Definition 14 (Direct Mapping). *The direct mapping is defined as a transducing mapping $\delta^{\texttt{next}}_{\iota_{direct},\epsilon_{direct}}$, where the function ι_{direct} maps every natural number i to an individual* `e`i*. The individuals* `match`i_j *uniquely identify a match inside a trace for the jth variable of the ith event, and we regard variables as strings containing their names. Function ϵ_{direct} is defined as follows:*

$$\begin{aligned}
&\epsilon_{direct}(\mathsf{ev}_i(\mathsf{r}, v_1 : t_1, \ldots, v_n, t_n)) = \\
&\ \{\texttt{Event}(\iota_{direct}(i)), \texttt{appliesRule}(\iota_{direct}(i), \mathsf{r})\} \cup \\
&\ \bigcup_{j \le n} \big(\{\texttt{match}(\iota_{direct}(i), \texttt{match}i_j), \texttt{var}(\texttt{match}i_j, v_j), \texttt{term}(\texttt{match}i_j, \eta(t_j))\} \cup \delta(t_j)\big)
\end{aligned}$$

where $\delta(t_j)$ deterministically generates the axioms for the tree structure of the term t_j according to Def. 3 and $\eta(t_j)$ returns the individual of the head of t_j.

The properties `match`, `var` and `term` connect each event with its parameters. For example, the match v : `layer0` of the first event in Ex. 4, generates

$$\texttt{match}(\texttt{e1}, \texttt{match0_1}), \texttt{var}(\texttt{match0_1}, \text{“v”}), \texttt{term}(\texttt{match0_1}, \texttt{layer0})$$

where `e1` is the representation of the event and `match0_1` is the representation of the match in the KB. The complete direct mapping is given in the following example.

Example 10. The direct mapping of Ex. 4 is as follows. We apply the ϵ_{direct} function to all three events, where each event has one parameter.

$$\Big\{\texttt{Event}(\texttt{e1}), \texttt{Event}(\texttt{e2}), \texttt{Event}(\texttt{e3}), \texttt{Next}(\texttt{e1}, \texttt{e2}), \texttt{Next}(\texttt{e2}, \texttt{e3}), \texttt{appliesRule}(\texttt{e1}, \text{“}\texttt{depositStego}\text{”}),$$
$$\texttt{appliesRule}(\texttt{e2}, \text{“}\texttt{erode}\text{”}), \texttt{appliesRule}(\texttt{e3}, \text{“}\texttt{depositTRex}\text{”}), \texttt{match}(\texttt{e1}, \texttt{m1}), \texttt{var}(\texttt{m1}, \text{“v”}),$$
$$\texttt{term}(\texttt{m1}, \texttt{layer0}), \texttt{match}(\texttt{e2}, \texttt{m2}), \texttt{var}(\texttt{m2}, \text{“v”}), \texttt{term}(\texttt{m2}, \texttt{layer0}), \texttt{match}(\texttt{e3}, \texttt{m3}),$$
$$\texttt{var}(\texttt{m3}, \text{“v”}), \texttt{term}(\texttt{m3}, \texttt{layer1})\Big\}$$

7 Discussion

Querying and Stability. The mapping can be used by the domain users to interact with the system. For one, it can be used to retrieve the result of the computation using the vocabulary of a domain. For example, the following SPARQL [44] query retrieves all depositions generated during the Jurassic:

```
SELECT ?l WHERE {?l a Layer. ?l during Jurassic}
```

Indeed, one of the main advantages of knowledge bases is that they enable ontology-based data access [46]: uniform data access in terms of a given domain. Another possibility is to use *justifications* [5]. Justifications are minimal sets of axioms responsible for entailments over a knowledge base, e.g., to find out why it is inconsistent. They are able to explain, during an interaction, why certain steps are not possible.

The programmers do not need to design a complete knowledge base – for many domains knowledge bases are available, for example in form of industrial standards [26,23]. For more specific knowledge bases, clear design principles based on experiences in ontology engineering are available [17]. Note that these KBs are stable and do rarely change. Our system requires a static domain knowledge, as changes in the DKDP can invalidate traces during execution without executing a rule, which is, thus, not a limitation if one uses stable ontologies.

The direct mapping uses a fixed vocabulary, but one can formulate the connection to the domain knowledge by using additional axioms. In Ex. 10, one can declare every event to be a layer. The axiom for `depositStego` is as follows.

$$\texttt{appliesRule}\ \textbf{value}\ \text{“}\texttt{depositStego}\text{”}\ \textbf{SubClassOf}\ \texttt{contains}\ \textbf{value}\ \texttt{Stegosaurus}$$

The exact mapping strategy is application-specific – for example, to remove information `erode` must be handled through additional axioms as well, for example by adding a special concept `RemovedLayer` that is defined as all layers that

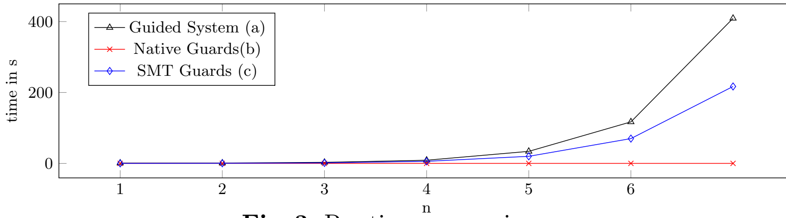

Fig. 3. Runtime comparison.

are matched on by some `erode` event. We next discuss some of the considerations when choosing the style of mapping, and the limitations of each.

There are, thus, two styles to connect trace and domain knowledge: One can add axioms connecting the vocabulary of traces with the vocabulary of the DKDP (direct mapping), or one can translate the trace into the vocabulary of the DKDP (first-order matching mapping, transducing mappings).

The two styles require different skills from the programmer to interact with the domain knowledge: The first style requires to express a trace as part of the domain as a set of ABox axioms, while the second one requires to connect *general* traces to the domain using TBox axioms. Thus, the second style operates on a higher level of abstraction and we conjuncture that such mappings may require more interaction with the domain expert and a deeper knowledge about knowledge graphs. However, the same insights needed to define the TBox axioms, are also needed to define the guards of a first-order matching mapping.

Naming Schemes. The transducing mappings and the first-order matching mapping have different naming schemes. A transducing mapping, and thus, a direct mapping, generate a new name *per event*, while the first-order matching mapping generates a fixed number of new names *per rule*: A transducing mapping can extract quite extensive knowledge from a trace, with the direct mapping giving a complete representation of it in a KB. As discussed, this requires the user to define general axioms. A first-order matching mapping must work with less names, and extract less knowledge from a trace. Its design requires to choose the right amount of abstraction to detect inconsistencies.

Evaluation. To evaluate whether the proposed system indeed gives a performance increase, we have implemented the running example[6] as follows: The system generates all traces up to length n, using three different transition systems: (a) The guided system (Def. 8) using the transducing mapping of Ex. 9. For reasoning, we use the Apache Jena framework [2]. (b) The guarded system (Def. 11) that uses a native implementation of the well-formedness predicate, and (c) the guarded system that uses the Z3 SMT solver [18] to check the first-order logic guards. The results are shown in Fig. 3.As we can see, the native implementation of the guarded systems is near instant for $n \leq 7$, while the guided

[6] https://github.com/Edkamb/KnowEnforce We slightly modified the example and replaced the asymmetry axioms by an equivalent formalization to fit the example into the fragment supported by the Jena OWL reasoner.

system needs more than 409s for $n = 7$ and shows the expected blow-up due to the `N2ExpTime`-completeness of reasoning in the logic underlying OWL [30]. The guarded system based on SMT similarly shows a non-linear behavior, but scales better then the guided system. For the evaluation, we ran each system for every n three times and averaged the numbers, using a Ubuntu 21.04 machine with an i7-8565U CPU and 32GB RAM. As we can see, the guarded system allows for an implementation that does not rely on an external, general-purpose reasoners to evaluate the guards and increases the scalability of the system, while the guided system does not scale even for small system and KBs.

8 Related Work

Runtime enforcement is a vast research field, for a recent overview we refer to the work of Falcone and Pinisetty [22], and give the related work for combinations of ontologies/knowledge bases and transitions systems in the following.

Concerning the combination of ontologies/knowledge bases and business process modeling, Corea et al. [16] point out that current approaches lack the foundation to annotate and develop ontologies together with business process rules. Our approach focuses explicitly on automating the mapping, or support developers in its development in a specific context, thus satisfying requirement 1 and 7 in their gap analysis for ontology-based business process modelling. Note that most work in this domain uses ontologies for the process model itself, similar to the ontology we give in Def. 13 and Def. 13 (e.g., Rietzke et al. [36]) or the current state (e.g., Corea and Delfmann [15]), not the trace. We refer to the survey of Corea et al. for a detailed overview.

Compared with existing simulators of hydrocarbon exploration [20,47], which formalized the domain knowledge of geological processes directly in the transition rules, we propose a *general* framework to formalize the domain knowledge in a knowledge base which is independent from the term rewriting system. This clear separation of concerns makes it easier for domain users to use the knowledge base for simulation without having the ability to program.

Tight interactions between programming languages, or transition systems, beyond logical programming and knowledge bases have recently received increasing research attention. The focus of the work of Leinberger [29,32] is the type safety of loading RDF data from knowledge bases into programming languages. Kamburjan et al. [28] semantically lift *states* for operations on the KB representation of the state, but are not able to access the trace. In logic programming, a concurrent extension of Golog [33] is extended to verify CTL properties with description logic assertions by Zarrieß and Claßen [48].

Cauli at al. [12] use knowledge bases to reason about the security properties of deployment configuration in the cloud, a high level representation of the overall system. As for traces, Pattipati et al. [34] introduce a debugger for C programs that operates on logs, i.e., special Traces. Their system operates post-execution and cannot guide the system. Al Haider et al. [1] use a similar technique to investigate logged traces of a program.

In runtime verification, knowledge bases has been investigated by Baader and Lippmann [6] in $\mathcal{ALC}$-LTL, which uses the description logic $\mathcal{ALC}$ instead of propositional variables inside of LTL. An overview over further temporalizations of description logics can be found in the work of Baader et al. [4]. Runtime enforcement has been using to temporal properties over traces since its beginnings [37], but, as a recent survey by Falcone and Pinisetty [22] points out, mainly for security/safety or usage control of libraries. In contrast, our work requires the enforcement to do any meaningful computation and uses a different way to express constraints than prior work: consistency with knowledge bases.

DatalogMTL extends Datalog with MTL operators [9,45] to enable ontology-based data access about sequences using inference rules. The ontology is expressed in these rules, i.e., it is not declarative but an additional programming layer, which we deem unpractical for domain users from non-computing domains. DatalogMTL has been used for queries [10] but not for runtime enforcement.

Traces have been explored from a logical perspective mainly in the style of CTL*, TLA and similar temporal logics. More recently, interest in more expressive temporal properties over traces of programming languages for verification using more complex approaches has risen and led to symbolic traces [11,19], integration of LTL and dynamic logics for Java-like languages [8] and trace languages based on type systems [27]. These approaches have in common that they aim for more expressive power and are geared towards better usability *for programmers* and simple verification calculi. They are only used for verification, not at runtime, and do not connect to formalized domain knowledge.

The guided system can be seen as a meta-computation, as put forward by Clavel et al. [14] for rewrite logics, which do not discuss the use of *consistency* as a meta-computation and instead program such meta computations explicitly.

9 Conclusion

We present a framework to use domain knowledge about dynamic processes to guide the execution of generic transition systems through runtime enforcement. We give a transformation to use of rule specific guards instead of using the domain knowledge directly as a consistency invariant over knowledge bases. The transformation is transparent and the domain user can interact with the system without being aware of the transformation or implementation details. To reduce the working load on the programmer, we discuss semi-automatic design of mappings using transducing approaches and a pre-defined direct mapping. We also discuss further alternatives, such as additional axioms on the events, and the use of local well-formedness predicates for certain classes of mappings.

Future Work. We plan to investigate how our system can interact with knowledge base evolution [24], a more declarative approach for changes in knowledge bases, as well as other approaches to modeling sequences in knowledge bases [40].

Acknowledgements This work was supported by University of Bergen and Research Council of Norway via `SIRIUS` (237898) and `PeTWIN` (294600).

References

1. N. Al Haider, B. Gaudin, and J. Murphy. Execution trace exploration and analysis using ontologies. In *RV*, volume 7186 of *LNCS*, pages 412–426. Springer, 2011.
2. Apache Foundation. Apache jena. https://jena.apache.org/.
3. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
4. F. Baader, S. Ghilardi, and C. Lutz. LTL over description logic axioms. *ACM Trans. Comput. Log.*, 13(3):21:1–21:32, 2012.
5. F. Baader and B. Hollunder. Embedding defaults into terminological knowledge representation formalisms. *J. Autom. Reason.*, 14(1):149–180, 1995.
6. F. Baader and M. Lippmann. Runtime verification using the temporal description logic ALC-LTL revisited. *J. Appl. Log.*, 12(4):584–613, 2014.
7. S. Baset and K. Stoffel. Object-oriented modeling with ontologies around: A survey of existing approaches. *Int. J. Softw. Eng. Knowl. Eng.*, 28(11-12):1775–1794, 2018.
8. B. Beckert and D. Bruns. Dynamic logic with trace semantics. In *CADE*, volume 7898 of *LNCS*, pages 315–329. Springer, 2013.
9. S. Brandt, E. G. Kalayci, R. Kontchakov, V. Ryzhikov, G. Xiao, and M. Zakharyaschev. Ontology-based data access with a horn fragment of metric temporal logic. In *AAAI*, pages 1070–1076. AAAI Press, 2017.
10. S. Brandt, E. G. Kalayci, V. Ryzhikov, G. Xiao, and M. Zakharyaschev. Querying log data with metric temporal logic. *J. Artif. Intell. Res.*, 62:829–877, 2018.
11. R. Bubel, C. C. Din, R. Hähnle, and K. Nakata. A dynamic logic with traces and coinduction. In *TABLEAUX*, volume 9323 of *LNCS*, pages 307–322. Springer, 2015.
12. C. Cauli, M. Li, N. Piterman, and O. Tkachuk. Pre-deployment security assessment for cloud services through semantic reasoning. In *CAV (1)*, volume 12759 of *LNCS*, pages 767–780. Springer, 2021.
13. K. L. Clark and F. G. McCabe. Ontology oriented programming in go! *Appl. Intell.*, 24(3):189–204, 2006.
14. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. Reflection, metalevel computation, and strategies. In *All About Maude*, volume 4350 of *LNCS*, pages 419–458. Springer, 2007.
15. C. Corea and P. Delfmann. Detecting compliance with business rules in ontology-based process modeling. In J. M. Leimeister and W. Brenner, editors, *Towards Thought Leadership in Digital Transformation: 13. Internationale Tagung Wirtschaftsinformatik, WI 2017, St.Gallen, Switzerland, February 12-15, 2017*, 2017.
16. C. Corea, M. Fellmann, and P. Delfmann. Ontology-based process modelling - will we live to see it? In A. K. Ghose, J. Horkoff, V. E. S. Souza, J. Parsons, and J. Evermann, editors, *Conceptual Modeling - 40th International Conference, ER 2021, Virtual Event, October 18-21, 2021, Proceedings*, volume 13011 of *Lecture Notes in Computer Science*, pages 36–46. Springer, 2021.
17. J. Davies, R. Studer, and P. Warren. *Semantic Web technologies: trends and research in ontology-based systems*. John Wiley & Sons, 2006.
18. L. M. de Moura and N. S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
19. C. C. Din, R. Hähnle, E. B. Johnsen, K. I. Pun, and S. L. T. Tarifa. Locally abstract, globally concrete semantics of concurrent programming languages. In *TABLEAUX*, volume 10501 of *LNCS*, pages 22–43. Springer, 2017.

20. C. C. Din, L. H. Karlsen, I. Pene, O. Stahl, I. C. Yu, and T. Østerlie. Geological multi-scenario reasoning. In *NIK*. Bibsys Open Journal Systems, Norway, 2019.
21. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects Comput.*, 27(3):551–572, 2015.
22. Y. Falcone and S. Pinisetty. On the runtime enforcement of timed properties. In *RV*, volume 11757 of *LNCS*, pages 48–69. Springer, 2019.
23. S. R. Fiorini, J. Bermejo-Alonso, P. J. S. Gonçalves, E. P. de Freitas, A. O. Alarcos, J. I. Olszewska, E. Prestes, C. Schlenoff, S. V. Ragavan, S. A. Redfield, B. Spencer, and H. Li. A suite of ontologies for robotics and automation [industrial activities]. *IEEE Robotics Autom. Mag.*, 24(1):8–11, 2017.
24. G. Flouris, D. Manakanatas, H. Kondylakis, D. Plexousakis, and G. Antoniou. Ontology change: classification and survey. *Knowl. Eng. Rev.*, 23(2):117–152, 2008.
25. M. Horridge, N. Drummond, J. Goodwin, A. L. Rector, R. Stevens, and H. Wang. The manchester OWL syntax. In *OWLED*, volume 216 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.
26. IEEE ORA WG. IEEE standard ontologies for robotics and automation. *IEEE Std 1872-2015*, pages 1–60, 2015.
27. E. Kamburjan. Behavioral program logic. In *TABLEAUX*, volume 11714 of *LNCS*, pages 391–408. Springer, 2019.
28. E. Kamburjan, V. N. Klungre, R. Schlatte, E. B. Johnsen, and M. Giese. Programming and debugging with semantically lifted states. In *ESWC*, volume 12731 of *LNCS*, pages 126–142. Springer, 2021.
29. E. Kamburjan and E. V. Kostylev. Type checking semantically lifted programs via query containment under entailment regimes. In *Description Logics*, volume 2954 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021.
30. Y. Kazakov. SRIQ and SROIQ are harder than SHOIQ. In *Description Logics*, volume 353 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
31. E. Kharlamov, F. Martín-Recuerda, B. Perry, D. Cameron, R. Fjellheim, and A. Waaler. Towards semantically enhanced digital twins. In *IEEE BigData*, pages 4189–4193. IEEE, 2018.
32. M. Leinberger. *Type-safe Programming for the Semantic Web*. PhD thesis, University of Koblenz and Landau, Germany, 2021.
33. H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997.
34. D. K. Pattipati, R. Nasre, and S. K. Puligundla. BOLD: an ontology-based log debugger for C programs. *Autom. Softw. Eng.*, 29(1):2, 2022.
35. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
36. E. Rietzke, R. Bergmann, and N. Kuhn. ODD-BP - an ontology- and data-driven business process model. In R. Jäschke and M. Weidlich, editors, *Proceedings of the Conference on "Lernen, Wissen, Daten, Analysen", Berlin, Germany, September 30 - October 2, 2019*, volume 2454 of *CEUR Workshop Proceedings*, pages 310–321. CEUR-WS.org, 2019.
37. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
38. SNOMED International. Snomed ct. https://www.snomed.org/.
39. A. Soylu, E. Kharlamov, D. Zheleznyakov, E. Jiménez-Ruiz, M. Giese, M. G. Skjæveland, D. Hovland, R. Schlatte, S. Brandt, H. Lie, and I. Horrocks. Optiquevqs: A visual query system over ontologies for industry. *Semantic Web*, 9(5):627–660, 2018.

40. E. D. Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intell. Syst.*, 24(6), 2009.
41. W. M. P. van der Aalst. Business process management: A comprehensive survey. *ISRN Software Engineering*, 2013:507984, Feb 2013.
42. W3C, OWL Working Group. Web ontology language. https://www.w3.org/OWL.
43. W3C, RDF Working Group. Resource description framework. https://www.w3.org/RDF.
44. W3C, SPARQL Working Group. Sparql 1.1 query language. https://www.w3.org/TR/sparql11-query/.
45. P. A. Walega, B. C. Grau, M. Kaminski, and E. V. Kostylev. Datalogmtl: Computational complexity and expressive power. In *IJCAI*, pages 1886–1892. ijcai.org, 2019.
46. G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyaschev. Ontology-based data access: A survey. In J. Lang, editor, *IJCAI 2018*, pages 5511–5519. ijcai.org, 2018.
47. I. C. Yu, I. Pene, C. C. Din, L. H. Karlsen, C. M. Nguyen, O. Stahl, and A. Latif. *Subsurface Evaluation Through Multi-scenario Reasoning*, pages 325–355. Springer International Publishing, Cham, 2021.
48. B. Zarrieß and J. Claßen. Verification of knowledge-based programs over description logic actions. In *IJCAI*, pages 3278–3284. AAAI Press, 2015.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.