Knowledge-driven Stream Monitor Synthesis

Åsmund Aqissiaq Arild Kløvstad[1], Riccardo Sieve[1], Andrea Pferscher[1], Eduard Kamburjan[1,2], Einar Broch Johnsen[1], and Martin Leucker[3]

[1] University of Oslo, Oslo, Norway
{aaklovst,riccasi,andreapf,einarj}@uio.no
[2] IT University of Copenhagen, Copenhagen, Denmark
eduard.kamburjan@itu.dk
[3] University of Lübeck, Lübeck, Germany
leucker@isp.uni-luebeck.de

Abstract. Runtime monitoring analyzes system traces based on a given specification to provide insights into complex system behavior. However, these specifications are typically defined at a low-level of abstraction and require expertise with the monitoring tool. Translating high-level system properties to correct combinations of monitor specifications, is error-prone and may require manual effort from both domain and runtime monitoring experts. This hampers the use of runtime monitoring for settings that depend on agility in specifications, e.g., in settings where users ask about system properties. To overcome this problem, we propose DynaMo, a synthesis method for stream monitors that bridges the gap between system-level properties and monitor specifications by leveraging formalized knowledge. DynaMo synthesizes and instruments a monitor specification given a knowledge base. We show how to guarantee soundness of all monitors derived this way, provide a prototype implementation of DynaMo for TeSSLa, and demonstrate its applicability on a case study in which users of a digital twin investigate marine ecosystems in the Oslo Fjord.

1 Introduction

Runtime monitoring is a lightweight formal method that analyzes system behavior by verifying whether system traces adhere to given formal specifications [23, 35]. By analyzing output behavior, runtime monitoring can be applied to black-box systems with multiple heterogeneous system components. Although runtime monitoring has traditionally targeted software and hardware systems [25], successful applications in aerospace [7, 33, 38], automotive [19, 48], and cyber-security [47] show that the research focus in runtime monitoring is increasingly shifting towards other application domains [41].

Many of us have experienced first-hand that it can be both challenging and time-consuming to correctly develop formal specifications for complex systems, even for seasoned formalists! Case studies also show that engineers have difficulty defining specifications for their systems [17, 26, 29], which impedes the application of formal methods, including runtime monitoring. To define specifications in

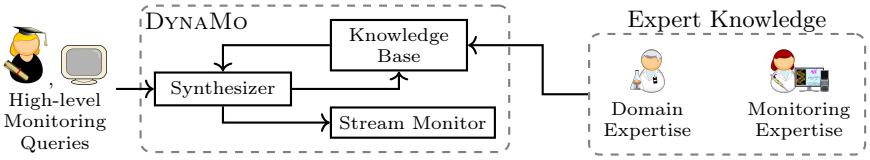


Fig. 1: Overview of the DynaMo approach.

complex temporal logic, approaches such as Sugar [8], SALT [6] and the temporal patterns of Dwyer et al. [17] propose syntactic sugar to simplify the definition of LTL or CTL specifications. While this line of work alleviates the low-level engineering of temporal formulas, runtime monitoring in domains beyond software additionally requires that domain knowledge is mapped onto the temporal logic formulas. Runtime monitoring with domain specific modules have been proposed [19] to provide parametric specifications tailored for a specific domain. However, this requires that domain experts and runtime monitoring experts jointly develop specifications *within the runtime monitoring framework*—any existing formalized knowledge has to be re-expressed in the monitoring framework.

To overcome this problem, we propose DynaMo, a method to automatically synthesize formal specifications that leverages formalized domain knowledge. DynaMo allows expert users or various other systems to monitor a target system by means of domain-specific queries (cf. Fig. 1). To enable the knowledge-driven synthesis of specifications, we extend knowledge bases that formalize reusable domain knowledge with atomic constraints that combine monitoring templates and system interfaces.

Our development is motivated by work on the *Oslo Fjord Digital Twin*[4] [31], which aims to monitor and evaluate ecological conditions in the Oslo Fjord.[5] The project includes 20 researchers and master students in the fields of oceanography, fluid mechanics, data science, computer science, and marine ecology, which demands effective knowledge sharing between domains. Domain and monitoring experts have different knowledge and formalisms, necessitating a common framework that combines the two. Knowledge bases allow the formalization of logic relations within complex systems. In the Oslo Fjord Digital Twin, this knowledge base is used to formalize the marine ecosystem and its relationship to physical environmental factors.

DynaMo extends knowledge bases with formalized domain knowledge with formalizations that enable the synthesis of runtime monitors based on user queries. The main contributions of this paper can be summarized as follows:

- DynaMo: a *knowledge-driven method to synthesize runtime monitors*, informed by both domain knowledge and monitoring knowledge (Sect. 3);
- A method to *prove soundness* for instances of the proposed method (Sect. 4);
- A prototype tool[6] implementing the DynaMo method (Sect. 5); and
- An evaluation of DynaMo on the Oslo Fjord Digital Twin (Sect. 6).

[4] https://ebjohnsen.org/project/oslofjord
[5] The Oslo Fjord is an inlet on the Skagerrak strait, south of the Norwegian capital.
[6] https://github.com/oslofjord-twin/oslofjord-dynamo.

2 Preliminaries

This section covers preliminary notions and definitions for the two key ingredients of DynaMo: stream-based runtime monitoring and knowledge bases.

Stream-based Runtime Monitoring. Stream-based runtime verification (SRV) [13, 15] considers runtime monitoring as a stream transformation problem. Stream specifications describe stream transformers, which operate on a number of input streams to produce a number of output streams. Sequences of events to monitor form the input stream(s); these are transformed to output streams acting as the monitoring result(s). Note that the outputs in SRV are not limited to Boolean streams; this allows enhanced analysis on outputs and the possibility of nesting multiple stream transformers.

We formalize stream-based runtime monitoring in terms of *stream expressions* and *stream specifications*. An *SRV language* consists of an expression language and its semantics. A *stream specification* in an SRV language defines a set of input streams and the computations needed to produce a set of output streams.

Definition 1 (SRV Languages). *Let X be a set of variable names and D a stream of values with $X \cap D = \emptyset$. An* SRV language *is a triple $(\mathbb{E}, D, \varepsilon)$, where (1) $\mathbb{E}$ is a set of stream expressions such that $\mathbb{E}(X)$ denotes expressions over X, and (2) $\varepsilon \colon \mathbb{E}(D) \to D$ is an evaluation function.*

The expression language $\mathbb{E}$ must be composable; i.e., nested expressions can be combined by a natural function $\mu \colon \mathbb{E}(\mathbb{E}(X)) \to \mathbb{E}(X)$. Given a variable assignment $\sigma \colon X \to D$, we obtain a valuation $[\![e]\!]^{\varepsilon}_{\sigma}$ *for $e \in \mathbb{E}(X)$ by lifting to expressions and composing with ε, denoted $[\![\_]\!]^{\varepsilon}_{\sigma} \colon \mathbb{E}(X) \xrightarrow{\mathbb{E}(\sigma)} \mathbb{E}(D) \xrightarrow{\varepsilon} D$.*

Definition 2 (Stream Specifications). *A* stream specification *in an SRV language $(\mathbb{E}, D, \varepsilon)$ is a tuple $(I, O, (e_x)_{x \in O})$, where I and O are sets of* input *and* output variables *with $I \cap O = \emptyset$, and $(e_x)_{x \in O}$ is an O-indexed set of expressions over variables; i.e., for each $x \in O$ there is an expression $e_x \in \mathbb{E}(I \cup O)$.*

The goal of the stream specification is to define a stream transformer that, given values for the input stream, uniquely and uniformly assigns values to the output streams. If this is possible, the specification is *well-defined*. Given valuations σ_X and σ_Y with disjoint domains X, Y, we write $\sigma_X \cup \sigma_Y$ for the unique valuation with domain $X \cup Y$ that coincides with σ_X on X and σ_Y on Y.

Definition 3 (Well-defined Specifications [34]). *A stream specification $\phi = (I, O, (e_x)_{x \in O})$ is* well-defined *if, for each input valuation $\sigma_I : I \to D$, there exists a unique output valuation $\sigma_O : O \to D$ such that, for every $x \in O$, we have $(\sigma_I \cup \sigma_O)(x) = [\![e_x]\!]^{\varepsilon}_{\sigma_I \cup \sigma_O}$. The* denotational semantics *of a well-defined specification is the function $[\![\phi]\!] \colon (I \to D) \to (O \to D)$ mapping input valuations to their unique output valuations.*

We only consider specifications that are *efficiently monitorable* [15]. Essentially, efficiently monitorable specifications are those that can be expressed by only referring to the past and present events.

Knowledge Bases. A knowledge base organizes information in a logical format and provides an interface to access this information. For simplicity, we here consider first-order logic (FOL) knowledge bases with two kinds of operations in the interface: One can either check the validity (or satisfiability) of the contained information, or retrieve certain answers [36,37] via queries.

Definition 4 (FOL Knowledge Bases). *Given a* signature $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$ *with sets of* constant- *(Σ_0),* predicate- *(Σ_1) and* relation-*symbols (Σ_2), and a set X of variables, with $X \cap \Sigma = \emptyset$. Let c range over Σ_0, P over Σ_1, R over Σ_2, and x over X.* Formulas ϕ *and* terms t *are defined as follows.*

$$\phi ::= P(t) \mid R(t,t) \mid \forall x.\phi \mid \neg\phi \mid \phi \wedge \phi \qquad\qquad t ::= x \mid c$$

A sentence *over Σ is first-order formula over Σ with no free variables, a* knowledge base $\mathcal{K}$ *over Σ is a set of sentences over Σ.*

The satisfiability of a formula ϕ by a *model* $\mathcal{M} = (V, \mathcal{I})$ and a *variable assignment* $\beta : X \to V$ is denoted $\mathcal{M}, \beta \models \phi$ and defined inductively in the usual way (e.g., [2]). A formula ϕ is *satisfiable* if there exists a model and variable assignment that satisfies ϕ. A formula ϕ is *valid*, denoted $\models \phi$, if it is satisfied by all models and assignments. Here V is a *value domain* and $\mathcal{I}$ is an *interpretation* that assigns (1) elements of V to constants, (2) subsets of V to predicates, and (3) subsets of $V \times V$ to relations. Terms have denotations $[\![x]\!]_{\mathcal{M},\beta} = \beta(x)$ for variables and $[\![c]\!]_{\mathcal{M},\beta} = \mathcal{I}(c)$ for constants.

Definition 5 (Knowledge Base Semantics). *A knowledge base $\mathcal{K}$ is* satisfiable *(resp.* valid*) if $\bigwedge_{\phi\in\mathcal{K}}$ is satisfiable (resp. valid). $\mathcal{K}$* entails *a sentence ψ, denoted $\mathcal{K} \models \psi$, if the formula $\models \bigwedge_{\phi\in\mathcal{K}} \phi \implies \psi$ is valid.*

A knowledge base is *consistent* iff it does not imply a contradiction, which in FOL is equivalent to satisfiability. Consistency is a primary concern for knowledge bases and is effectively checked by knowledge base software.

A *first-order query* is a first-order formula with free variables. Its answers are tuples of constants such that, if the constants are assigned to the free variables, then the resulting first-order sentence is entailed by the knowledge base.

Definition 6 (Queries). *Let $\mathcal{K}$ be a knowledge base over a signature Σ and denote by $\phi(x_1, \ldots, x_n)$ a formula with n free variables. Then ϕ is a* query *and its* certain answer set *is defined by:*

$$[\![\phi(x_1, \ldots, x_n)]\!]_{\mathcal{K}} = \left\{(c_1, \ldots, c_n) \in \Sigma_0^n \mid \mathcal{K} \models \phi(c_1, \ldots, c_n)\right\} .$$

3 Stream Monitor Synthesis with DynaMo

DynaMo is a method to dynamically synthesize monitors from knowledge bases that contain both domain expertise and monitoring expertise. While the knowledge base for a given use case depends on the domain and used SRV language, the synthesizer uses a *core knowledge base* (cf. Fig. 2) as an interface. We now present this core knowledge base and the process for synthesizing monitors from monitor queries.

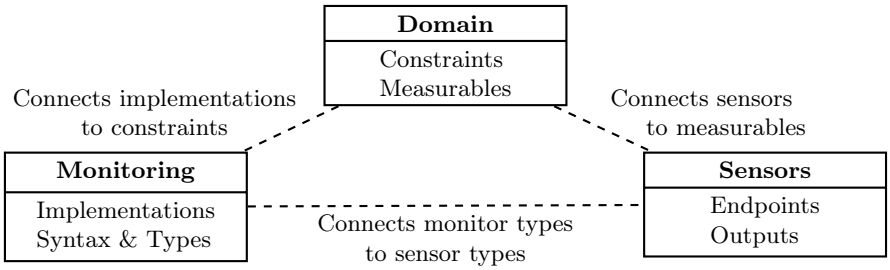


Fig. 2: Components of the DynaMo core knowledge base.

3.1 Formalizing Stream Monitoring in a Knowledge Base

DynaMo's synthesizer extracts information from the knowledge base using first-order queries. As the knowledge base depends on the use case, i.e., the domain of the entities to be monitored, and the SRV, i.e., the target language of the synthesizer, we require a core knowledge base that contains (a) fixed vocabulary (shown in `typewriter`-font) to formulate generic queries, and (b) constraints on this vocabulary. For each use case, the domain and monitoring expertise must be encoded by extending the core knowledge base while respecting these constraints.

The core knowledge base, illustrated in Fig. 2, has three parts: domain (a), monitoring (b), and sensor (c). The domain part (a) describes core terminology about what can be monitored in principle. The monitoring part contains monitoring expertise about a certain SRV w.r.t. the domain, and the sensor part contains knowledge about sensors, how they connect to the domain and how to retrieve input streams for the runtime monitors from them. In the following we use the term *notion* to refer to a signature symbol with an intended meaning. A fully formal representation with additional properties is available in the auxiliary material [1].

The *domain core knowledge base* introduces two notions: *measurables* and *constraints*. A measurable is a *quality* of a digital or physical system that can be measured, e.g., the temperature at a given location or the bandwidth of a given connection. A measurable is not a measurement. A *constraint* on a measurable is an abstract condition on this quality, e.g., a maximal temperature.

The *sensor core knowledge base* introduces notions of *sensors* and *streams*. Sensors measure measurables; for example, a thermometer measures temperature, and a wind speed gauge measures wind speed. The data from a sensor is provided as a stream of a given (XSD [10]) type, connected to a stream language type by `corresponds`. The sensor core knowledge base employs the established SAREF [21] and DCAT [14] vocabularies.

The *monitoring core knowledge base* describes monitor implementations using *templates*: stream expressions in the SRV language with typed parameters. Templates are connected to the atomic constraints they implement, and parameters instantiated with constant values or streams when constructing monitors.

In addition to the core knowledge base, DynaMo relies on an *instantiation* of necessary domain and monitoring knowledge. Modular instantiations mean the different parts are reusable: the SRV expertise can be used in multiple domains.

A domain knowledge base introduces domain specific concepts and qualities, and a set of mapping sentences connects it to concepts in the core. A monitoring knowledge base contains templates implementing building blocks of specifications as well as the available types. Types must be connected to corresponding sensor output types, and templates to the constraints they implement. Note that one template may implement more than one constraint.

Definition 7 (Knowledge Base Instantiation). *Given a core knowledge base $\mathcal{K}$ with signature Σ, a* knowledge base instantiation *is a four-tuple of knowledge bases $(\mathcal{K}_{\mathsf{domain}}, \mathcal{K}_{\mathsf{map}}, \mathcal{K}_{\mathsf{srv}}, \mathcal{K}_{\mathsf{impl}})$ where*

- *$\mathcal{K}_{\mathsf{domain}}$ is a domain knowledge base with signature Σ_{domain} disjoint from Σ defining domain-specific concepts and qualities,*
- *$\mathcal{K}_{\mathsf{map}}$ is a knowledge mapping over $\Sigma_{\mathsf{domain}} \cup \Sigma$ connecting domain knowledge to the core concepts,*
- *$\mathcal{K}_{\mathsf{srv}}$ contains implementation templates and types in the chosen SRV language, and*
- *$\mathcal{K}_{\mathsf{impl}}$ is a set of sentences $\texttt{implements}(\texttt{x}, \texttt{y})$ connecting implementations to atomic constraints.*

The fully instantiated knowledge base is the union of these, denoted

$$\mathcal{K}^{+} = \mathcal{K}_{\mathsf{domain}} \cup \mathcal{K}_{\mathsf{srv}} \cup \mathcal{K}_{\mathsf{map}} \cup \mathcal{K}_{\mathsf{impl}} \cup \mathcal{K} \ .$$

Example 1 (Oslo Fjord Digital Twin Knowledge Base). We consider a simplified version of the knowledge base of the Oslo Fjord Digital Twin as a running example, informally introducing concepts as needed. A formal account is given in Sect. 6.

For this knowledge base, $\mathcal{K}_{\mathsf{domain}}$ includes qualities of temperature, salinity and cod observations. It also contains the concepts of good cod habitats, preferred water temperatures and maximal salinity. $\mathcal{K}_{\mathsf{map}}$ states that the qualities are measurable, that preferred temperature and salinity are atomic constraints, and that being a good habitat is a conjunction of these. The $\mathcal{K}_{\mathsf{srv}}$ has templates with typed parameters implementing ranges (float), maxima (float), and observations (bool), that are mapped to the atomic constraints. Additionally, the knowledge base contains temperature and salinity sensors and a camera that can detect observations of cod. The temperature and salinity sensors provide streams of XSD decimals (corresponding to float), while the camera provides a stream of Booleans (corresponding to bool) indicating whether a cod was observed.

3.2 Synthesis of Stream Specifications

Synthesis is triggered by a *monitor query* and results in a fully instrumented runtime monitor. We construct a specification with measurables as input variables,

constraints as output variables with defining expressions given by monitoring templates, and an additional designated output variable for the monitor query. We describe the process in three steps: (1) determining the required input and output variables, (2) producing a defining stream expression for each output variable, and (3) translating the query to a stream expression. Finally we combine these parts into a specification and instrument it with data streams.

Monitor queries are the starting point of synthesis. The definition is general, allowing for a wide range of languages—so long as they allow a translation into stream expressions. In Sect. 6 we show a concrete example.

Definition 8 (Monitor Queries). *Let $\mathcal{Q}(X)$ denote the set of monitor queries over variables X, and $[\_]\colon \mathcal{Q}(X) \to \mathbb{E}(X)$ a translation into stream expressions.*

Let the set of variables occurring in a query $q \in \mathcal{Q}(X)$ be denoted $vars(q) \subseteq X$. In practice, the input to the monitor synthesis will be a query over constraints defined in the knowledge base.

Example 2 (Oslo Fjord Digital Twin Queries). Consider as a user a marine biologist researching Atlantic cod. She wonders whether standard assumptions about cod habitats are valid for observed specimen. Using connectives defined in the query language, she formulates the query `codObserved=>goodCodHabitat`.

Given a set of constraints $\mathcal{C}$ and a query $q \in \mathcal{Q}(\mathcal{C})$, determining the set of input variables requires unfolding q into a *constraint expression*. Constraint expressions allow for monitors that combine atomic properties into more complex constraints. Like queries, constraint expressions must also be translatable to stream expressions.

Definition 9 (Constraint Expressions). *Let $\mathbb{C}(X)$ denote the set of constraint expressions over variables X, and $(\!|\_|\!)\colon \mathbb{C}(X) \to \mathbb{E}(X)$ a translation function from constraint expressions to stream expressions.*

Constraint expressions can be encoded in a knowledge base as a directed graph connected by the `comprises`-relation. For an operator $c \in \mathbb{C}$ and constraint expressions e, e_1, e_2, the sentences $\texttt{comprises}(e, e_1)$, $\texttt{comprises}(\texttt{e}, \texttt{e}_2)$ and $\texttt{hasT}(e, c)$ encode an expression $e = c(e_1, e_2, \ldots)$. An atomic constraint relates a measurable to a constant. An *indirect* constraint is a reference (by name) to another constraint. Axioms enforce that the resulting graph is *acyclic*; i.e., the graph forms a tree with the starting constraint at the root and *atomic properties* as its leaves.

The query and constraint languages, along with their translation mappings, form a *language instantiation* used for specification synthesis.

Definition 10 (Language Instatiation). *A* language instantiation *is a pair of query- and constraint-languages with translation mappings $\mathcal{L} = ((\mathcal{Q}, [\_]), (\mathbb{C}, (\!|\_|\!)))$.*

$$conExpr_{\mathcal{K}}(c) = \begin{cases} (\texttt{Atomic}\ c) & \text{if } \mathcal{K} \models \texttt{constraintType}(c, \texttt{AtomicConstraint}) \\ (\texttt{Indirect}\ p) & \text{if } \mathcal{K} \models \texttt{constraintType}(c, \texttt{IndirectConstraint}) \\ & \quad \wedge\ \texttt{target}(c, p) \\ C[\{conExpr_{\mathcal{K}}(m') \mid m' \in [\![\texttt{comprises}(m, x)]\!]\}] & \\ & \text{if } \mathcal{K} \models \texttt{constraintType}(c, C) \end{cases}$$

$$streams_{\mathcal{K}}(x) = \begin{cases} \{m \mid (m, s) \in [\![\texttt{serviceFor}(m, s)]\!]\} & \text{if } x = (\text{Atomic } c) \\ & \quad \text{and } \mathcal{K} \models \texttt{verifies}(c, m) \\ streams_{\mathcal{K}}(conExpr_{\mathcal{K}}(c)) & \text{if } x = (\texttt{Indirect}\ c) \\ \{streams_{\mathcal{K}}(c) \mid c \in X\} & \text{if } x = C[X] \end{cases}$$

Fig. 3: Functions used in the synthesis.

$$\begin{aligned} \texttt{constraintType}(x, t) &:= \texttt{hasT}(x, t) \wedge \texttt{subType}(t, \texttt{Constraint}) \\ \texttt{implementation}(x, t) &:= \texttt{implements}(i, x) \wedge \texttt{functionName}(i, t) \\ \texttt{parameters}(c, p, v) &:= \texttt{implements}(i, c) \wedge \texttt{hasParam}(i, p') \\ &\quad \wedge\ \texttt{paramName}(p', p) \wedge \texttt{hasProp}(c, p, v) \\ \texttt{serviceFor}(m, s) &:= \texttt{observes}(x, m) \wedge \texttt{provides}(x, s) \end{aligned}$$

Fig. 4: Knowledge base queries for monitor synthesis.

The function *conExpr* (see Fig. 3) unfolds a constraint into an expression over atomic properties and constraint names in $\mathcal{C}$ by recursively querying for `constraintType` and `comprises`. This function communicates with a knowledge base $\mathcal{K}$ using the queries shown in Fig. 4.

Example 3 (Oslo Fjord Digital Twin Constraint Expressions). Recall from Example 1 that the knowledge base of the Oslo Fjord Digital Twin defines good cod habitats as a conjunction of atomic properties. The properties from the query in Example 2 get unfolded into the constraint expressions `Atomic(codObserved)` and `Conjunction(Atomic(codTemperature), Atomic(codSalinity))`.

The function $streams\colon \mathbb{C}(\mathcal{C}) \to M$, where M is the set of measurables (see Fig. 3), obtains the set of necessary input streams from these constraints, using the query `serviceFor`. We derive defining expressions by applying $(\!|\_|\!)$ from the translation mapping $\mathcal{L}$ and replacing each atomic property by its implementation. Combining these steps, we construct a defining expression directly from a constraint:

$$streamExpr^{\mathcal{L}}_{\mathcal{K}}\colon \mathcal{C} \xrightarrow{conExpr_{\mathcal{K}}} \mathbb{C}(\mathcal{P} \cup \mathcal{C}) \xrightarrow{(\!|\_|\!)} \mathbb{E}(\mathcal{P} \cup \mathcal{C}) \xrightarrow{(*)} \mathbb{E}(\mathbb{E}(I \cup \mathcal{C})) \xrightarrow{\mu} \mathbb{E}(I \cup \mathcal{C}).$$

Here, the step $(*)$ replaces atomic properties with stream expressions using `implementation` and `parameters` (Fig. 4), and $\mathcal{P}$ is a set of atomic properties.

Example 4 (Oslo Fjord Digital Twin Stream Expressions). The Oslo Fjord $\mathcal{K}_{\mathsf{domain}}$ contains sentences mapping `codTemperature` to a temperature range between 5.5 and 18 degrees and `codSalinity` to a maximum of 35.6, the $\mathcal{K}_{\mathsf{impl}}$ knowledge base maps these to SRV expressions, and $(\!|\_|\!)$ maps `Conjunction` to an infix operator &&. The first constraint expression in Example 3 gets translated to a stream expression as follows:

$$\begin{aligned}&\texttt{Conjunction}(\texttt{Atomic codTemperature}, \texttt{Atomic codSalinity})\\ &\xmapsto{(\!|\_|\!)} \texttt{codTemperature \&\& codSalinity}\\ &\xmapsto{\mu \circ (*)} \texttt{inRange}(\texttt{Temp}, 5.5, 18) \;\&\&\; \texttt{Sal} \leq 35.6\end{aligned}$$

Finally, we obtain a stream specification. The complete set of inputs is the union over all atomic constraints that occur in the stream expressions. The outputs are the constraints that occur in the monitor query, plus an additional stream `ok`, which is the stream used to answer the query.

Definition 11 (Stream Specification). *Given a fully instantiated knowledge base $\mathcal{K}^+$ with constraints $\mathcal{C}$ and language instantiation $\mathcal{L} = ((\mathcal{Q}, [\_]), (\mathbb{C}, (\!|\_|\!)))$, the* synthesized stream specification $\mathbf{spec}^{\mathcal{L}}_{\mathcal{K}^+}(q)$ *for a monitor query $q \in \mathcal{Q}(\mathcal{C})$ is the stream specification $(I, O, (e_x)_O)$ such that*

$$I = \bigcup_{c \in vars(q)} \{s \mid s \in streams_{\mathcal{K}^+}(c)\} \qquad O = vars(q) \cup \{\texttt{ok}\}$$
$$(e_x)_O = \{e_c = streamExpr^{\mathcal{L}}_{\mathcal{K}^+}(c) \mid c \in vars(q)\} \cup \{e_{\texttt{ok}} = [q]\}$$

Example 5 (Oslo Fjord Digital Twin Stream Specifications). Recall the query `codObserved=>goodCodHabitat` given in Example 2. We obtain the following stream specification from the query:

$$\begin{aligned}&I = \{\texttt{Temp}, \texttt{Sal}, \texttt{CodObservation}\}\\ &O = \{\texttt{codObserved}, \texttt{goodCodHabitat}, \texttt{ok}\}\\ &e_{\texttt{codObserved}} = \texttt{CodObservation}\\ &e_{\texttt{goodCodHabitat}} = \texttt{inRange}(\texttt{Temp}, 5.5, 18) \;\&\&\; \texttt{Sal} \leq 35.6\\ &e_{\texttt{ok}} = \texttt{!codObserved||goodCodHabitat}\end{aligned}$$

The specification can then be instrumented by connecting to the sensor endpoints for salinity, temperature, and cod observations.

4 Soundness of DynaMo

Synthesis is not guaranteed to succeed for all knowledge base and language instantiations $(\mathcal{K}_{\mathsf{domain}}, \mathcal{K}_{\mathsf{map}}, \mathcal{K}_{\mathsf{srv}}, \mathcal{K}_{\mathsf{impl}})$ and $(\mathcal{Q}, [\_], \mathbb{C}, (\!|\_|\!))$. We now consider consistency conditions on the instantiations that guarantee success. There are three kinds of consistency conditions, each targeting one interaction of the interface: *Logical consistency* ensures that validity and entailment queries are handled correctly, *data consistency* that retrieval queries are handled correctly and *language consistency* that the semantics of the SRV is respected. Logical consistency captures the notion that all provided knowledge bases are logically consistent.

$$\forall \mathtt{t} \in \Sigma_0^+ .\ \mathcal{K}^+ \models \mathtt{OntologyType(t)} \rightarrow \left| [\![\mathtt{corresponds(t}, t') \wedge \mathtt{MonitorType}(t')]\!]_{\mathcal{K}^+} \right| = 1 \tag{A1}$$

$$\forall \mathtt{p} \in \Sigma_0^+ .\ \mathcal{K}^+ \models \mathtt{AtomicProperty(p)} \rightarrow \left| [\![\mathtt{implements(p}, t') \wedge \mathtt{Template}(t')]\!]_{\mathcal{K}^+} \right| = 1 \tag{A2}$$

$$\forall \mathtt{t}, \mathtt{p} \in \Sigma_0^+ .\ \mathcal{K}^+ \models \mathtt{AtomicProperty(p)} \wedge \mathtt{Template(t)} \rightarrow [\![Q_1(\mathtt{t}, \mathtt{p})]\!]_{\mathcal{K}^+} = 1 \tag{A3}$$

$$\forall \mathtt{l}, \mathtt{m} \in \Sigma_0^+ .\ \mathcal{K}^+ \models \mathtt{Measurable(m)} \rightarrow \left| [\![\mathtt{Sensor}(s) \wedge \mathtt{measures}(s, \mathtt{m})]\!]_{\mathcal{K}^+} \right| = 1 \tag{A4}$$

$$\forall \mathtt{o}, \mathtt{n}, \mathtt{t}, \mathtt{t}' \in \Sigma_0^+ .\ \mathcal{K}^+ \models \mathtt{Parameter(o)} \wedge \mathtt{Name(n)} \wedge \mathtt{MonitorType(t)} \wedge \mathtt{t}' \in [\![Q_2(\mathtt{o}, \mathtt{n}, \mathtt{t})]\!]_{\mathcal{K}^+} \rightarrow \mathcal{K}^+ \models \mathtt{corresponds(t, t')} \tag{A5}$$

$$\forall \mathtt{p}, \mathtt{n}, \mathtt{t}, \mathtt{t}' \in \Sigma_0^+ .\ \mathcal{K}^+ \models \mathtt{Parameter(o)} \wedge \mathtt{Name(n)} \wedge \mathtt{MonitorType(t)} \wedge \mathtt{t}' \in [\![Q_3(\mathtt{p}, \mathtt{n}, \mathtt{t})]\!]_{\mathcal{K}^+} \rightarrow \mathcal{K}^+ \models \mathtt{corresponds(t, t')} \tag{A6}$$

$$\mathcal{K}^+ \not\models \nexists t .\ \mathtt{comprises}(t, t) \tag{A7}$$

where

$$\begin{aligned} Q_1 &= \mathtt{implements(t, p)} \wedge \mathtt{hasParam(t}, a) \wedge \mathtt{hasDataProp(p}, a, v) \\ Q_2 &= \exists i, c, v .\ \mathtt{Constraint}(c) \wedge \mathtt{Template}(i) \wedge \mathtt{implements}(i, c) \\ &\quad \wedge \mathtt{hasParam}(i, \mathtt{p}) \wedge \mathtt{hasDataProp}(c, \mathtt{n}, v) \wedge \mathtt{hasT}(v, t') \\ Q_3 &= \exists i, c, m, s, s' .\ \mathtt{Constraint}(c) \wedge \mathtt{Measurable}(m) \wedge \mathtt{Sensor}(s) \wedge \mathtt{Template}(i) \\ &\quad \wedge \mathtt{Stream}(s') \wedge \mathtt{implements}(i, c) \wedge \mathtt{verifies}(c, m) \\ &\quad \wedge \mathtt{measures}(s, m) \wedge \mathtt{provides}(s, s') \wedge \mathtt{offers}(s, t') \end{aligned}$$

Fig. 5: Consistency conditions for data consistency. To clearly distinguish the meta-logic of the conditions and the used FOL logic, the logical operators of the meta-logic are bold and blue.

Definition 12 (Logical Consistency). *Recall that $\mathcal{K}$ is the core knowledge base defined in Sect. 3.1. A knowledge base instantiation* $(\mathcal{K}_{\mathsf{domain}}, \mathcal{K}_{\mathsf{map}}, \mathcal{K}_{\mathsf{srv}}, \mathcal{K}_{\mathsf{impl}})$ *is* logically consistent *if $\mathcal{K}_{\mathsf{domain}} \cup \mathcal{K}_{\mathsf{map}} \cup \mathcal{K}_{\mathsf{srv}} \cup \mathcal{K}_{\mathsf{impl}} \cup \mathcal{K}$ is consistent.*

Knowledge base queries return logical constants, so we must not only ensure that the interpretations of $\mathcal{K}_{\mathsf{domain}} \cup \mathcal{K}_{\mathsf{map}} \cup \mathcal{K}_{\mathsf{srv}} \cup \mathcal{K}_{\mathsf{impl}} \cup \mathcal{K}$ are correct, but also that the data is present for retrieval; i.e., enough constants must be introduced. This is captured by *consistency conditions* on the signature and knowledge base.

Definition 13 (Data Consistency). *Let $(\mathcal{K}_{\mathsf{domain}}, \mathcal{K}_{\mathsf{map}}, \mathcal{K}_{\mathsf{srv}}, \mathcal{K}_{\mathsf{impl}})$ be a logically consistent instantiation of a core knowledge base K, $\mathcal{K}^+ = \mathcal{K}_{\mathsf{domain}} \cup \mathcal{K}_{\mathsf{map}} \cup \mathcal{K}_{\mathsf{srv}} \cup \mathcal{K}_{\mathsf{impl}} \cup \mathcal{K}$ the union of knowledge bases, and Σ_0^+ the constants in the unified signature. The instantiation is* data consistent *if the conditions in Fig. 5 hold.*

Condition A1 ensures that for each ontology type there is exactly one monitor type, i.e., each type provided by endpoints is uniquely mapped to a type of the SRV language; A2 that there is exactly one SRV template for each atomic property; A3 that constants exist for the parameters of each template; A4 that there is a sensor for each measurable; A5 and A6 encode detailed conditions on the existence of types; and A7 ensures that constraints are acyclic.

For data consistent instantiations, it is possible to synthesize runtime monitors with well-formedness properties.

Theorem 1 (Sound Instantiations). *Let $\mathcal{K}^+$ be a fully instantiated knowledge base and $\mathcal{L}$ a language instantiation. If the knowledge base instantiation is data consistent, then for all user queries q, the* synthesized stream specification $\mathbf{spec}^{\mathcal{L}}_{\mathcal{K}^+}(q)$ *(1) exists and is (2) syntactically correct, (3) well-typed and (4) well-defined.*

Proof (Point 4). The proof amounts to showing that each knowledge base query used in Def. 11 returns an answer, i.e. that their answer sets are non-empty. For `implementation`, `verifies`, `serviceFor`, we further require the answer to be unique, i.e. the answer set is a singleton.

By Eq. (A2), there is exactly one implementation for each atomic property, so `implementation` is sound. When querying `parameters`, we are in the atomic property case, so Equation (A3) ensures that the appropriately named parameters exist. Similarly, `streams` queries for `serviceFor` only in the atomic case, so Eq. (A4) ensures the existence of sensors at all locations. The remaining stream queries are sound by functionality constraints.

Equations (A5) and (A6) ensure that all parameters are correctly typed, while Equation (A7) ensures that no output stream computation relies on itself, which guarantees well-definedness.

To ensure correctness of the synthesized monitor, we need to reason about the SRV semantics: the translation to stream expressions must preserve the meaning of both monitor queries and constraints. This requires both the query and constraint language to have a semantics, and the domain to be comparable to that of the stream expression language.

Definition 14 (Sound Mapping). *Given an SRV $(\mathbb{E}, V, \varepsilon)$, let F be some arbitrary signature with semantics $\varepsilon_F \colon F(V) \to V$, and $[\_]_F$ a translation function mapping into $\mathbb{E}$. The mapping $[\_]_F$ is* semantically sound for ε_F *if $\varepsilon_F(t) = \varepsilon([t]_F)$ for all $t \in F(V)$.*

Soundness extends to variable substitutions by first replacing variables, and guarantees the same result whether an expression is evaluated before or after the translation.

Given a knowledge base $\mathcal{K}$ mapping atomic properties $\mathcal{P}$ to SRV expressions over input variables, a valuation $\sigma \colon I \to V$ assigns a value to each property by retrieving and then evaluating the implementation. For a property p, denote this value $[\![p]\!]^{\mathcal{K}}_{\sigma}$. In the statement of the following consistency theorem we will tacitly identify this function with its lifting to $\mathcal{Q}(\mathbb{C}([\![\_]\!]^{\mathcal{K}}_{\sigma})) \colon \mathcal{Q}(\mathbb{C}(\mathcal{P})) \to \mathcal{Q}(\mathbb{C}(V))$ to de-clutter notation, and similarly for direct evaluation.

Theorem 2 (Language Consistency). *Let $\mathcal{K}^+$ be a data-consistent fully instantiated knowledge base, $\mathcal{L} = ((\mathcal{Q}, [\_]), (\mathbb{C}, (\!|\_|\!)))$ a language instantiation and $\varepsilon_{\mathcal{Q}}$, $\varepsilon_{\mathbb{C}}$ evaluation functions for queries and constraints. If $[\_]$, $(\!|\_|\!)$ are semantically*

sound mappings for $\varepsilon_{\mathcal{Q}}$ and $\varepsilon_{\mathbb{C}}$ respectively, then for all queries q and valuations $\sigma\colon I \to V$, we have

$$[\![\mathbf{spec}^{\mathcal{L}}_{\mathcal{K}^+}(q)]\!](\sigma)(\mathtt{ok}) = \varepsilon_{\mathcal{Q}}\left(\left[\!\left[[\![(\mathcal{Q}(\mathit{conExpr}_{\mathcal{K}^+})(q))]\!]^{\mathcal{K}^+}_{\sigma}\right]\!\right]^{\varepsilon_{\mathbb{C}}}_{\sigma}\right) .$$

Proof (Sketch). The proof amounts to unfolding definitions and applying language consistency. On the left-hand side, we know from Theorem 1 that the specification exists and is well-defined. The result of applying it to σ and $\mathtt{ok}$ is $[\![e_{\mathtt{ok}}]\!]^{\varepsilon}_{\sigma}$, whose expression comes from $[q]$. On the right-hand side, $\mathcal{Q}(\mathit{conExpr}_{\mathcal{K}^+})$: $\mathcal{Q}(\mathcal{C}) \to \mathcal{Q}(\mathbb{C}(\mathcal{P}))$ unfolds constraint names to constraint expressions before $[\![\_]\!]^{\mathcal{K}^+}_{\sigma} : \mathcal{Q}(\mathbb{C}(\mathcal{P})) \to \mathcal{Q}(\mathbb{C}(V))$ evaluates atomic properties. Finally, evaluation functions for constraints and queries are used to evaluate the resulting expression. The initial unfolding is equivalent to the translations performed by $\mathbf{spec}^{\mathcal{L}}_{\mathcal{K}^+}$, and language consistency ensures that the evaluations agree.

5 Implementation and Practical Considerations

DynaMo integrates and builds on available knowledge bases, SRVs and frameworks to check consistency. While SRVs are readily available, e.g. LOLA [15,18] and TeSSLa [13], and language consistency is supported by programming language theory, practical considerations about knowledge bases can be found in the form of *knowledge graphs* [28]. We now discuss how knowledge graphs and ontologies enable our approach as one possible way to realize knowledge bases. For SRV, our implementation is exemplified on a partial instantiation for TeSSLa.

Knowledge Graphs. Knowledge graphs are structures for knowledge representations with a logical interpretation as description logics [3]. In this interpretation, a knowledge graph is a pair of (1) a set of terminological axioms that express general knowledge, and (2) a set of assertions that express concrete knowledge about individuals. Knowledge graphs can be implemented by a technology stack of the RDF data model, the OWL ontology language, and the SPARQL query language [27]. The operations needed for knowledge bases and data consistency are provided by knowledge graphs, e.g., querying with enabled reasoning. Note that our core knowledge base uses only binary relation symbols and can thus be represented as a graph. Thus, the core knowledge base forms an OWL ontology.

Implementing DynaMo with knowledge graphs have some further benefits. First, description logics are sub-logics of first-order logics where reasoning about logical consistency is decidable [3]. Second, knowledge graphs formalize *ontologies*; thus, the domain and sensor knowledge bases need not be designed from scratch, but can build on existing ontologies. Third, ontologies are supported by methodologies for domain modeling, e.g., the linked open terms (LOT) [39] used for the core knowledge base, and ontology mapping. Ontology matching and alignment are tasks to (automatically) connect ontologies that overlap in the modeled domain, but not in terminology [42]. This is exactly the work needed to instantiate and give mapping axioms between the different knowledge bases.

Implementation. Our implementation of DynaMo is available in auxiliary material [1], including the core knowledge base, a synthesizer instantiated for TeSSLa and a monitoring knowledge base for TeSSLa. The *core knowledge base* was developed using best practices from ontology engineering, following the specification and implementation steps of the LOT methodology [39]. It was scoped and requirements formulated as competency questions [9,32]: queries the ontology must be able to answer. These questions were used to automatically validate the ontology whenever it was modified during development. Our core knowledge base also profited from ontology reuse, specifically the SAREF [21] ontology and DCAT [14] vocabulary for IoT devices for both the domain and sensor cores, and the xsd data definitions [10]. Instantiations of the SRV knowledge base should follow fixed ontology patterns [20], which can be automated using, e.g., OTTR [44] or RML [16].

The *synthesizer* was implemented as a command-line tool with approximately 1000 lines of Haskell. It takes a query as input and connects to a SPARQL-endpoint to communicate with the knowledge base. The output is a TeSSLa specification and a list of necessary input streams and their endpoint URLs. Changing the query and SRV requires changing and recompiling the tool, but the design is modular to facilitate this usage. When instantiating the synthesizer with TeSSLa, the templates exposed in $\mathcal{K}_{\mathsf{srv}}$ constitute a library of stream functions over floating point data, including equality checks with a given epsilon, explicit range checks, and averaging over a variable sliding window. From this library, we manually generate function declarations to insert correct data into the knowledge base. When executing a monitor, Rust compiles the specification into a binary.

6 Case Study Evaluation

We present a case study to evaluate DynaMo and answer our research questions:

RQ1 Can the DynaMo method be instantiated for an existing system within the limits of knowledge graph expressivity?
RQ2 Can the consistency conditions be practically proven for an instantiation?
RQ3 How does the synthesis of specifications and their monitoring perform?

DynaMo is instantiated with the *Oslo Fjord Digital Twin* as a component to dynamically generate monitors for the living conditions of different fish species based on knowledge about their preferred ecological niches, such as preferred temperatures. The user asks a question about a species and selects a location in the fjord, and the system then synthesizes a monitor to answer with live data. For implementation details, see [1].

6.1 Instantiation

Fully instantiating DynaMo requires a monitoring query language, the mapping knowledge bases and that we ensure consistency of the overall instantiation.

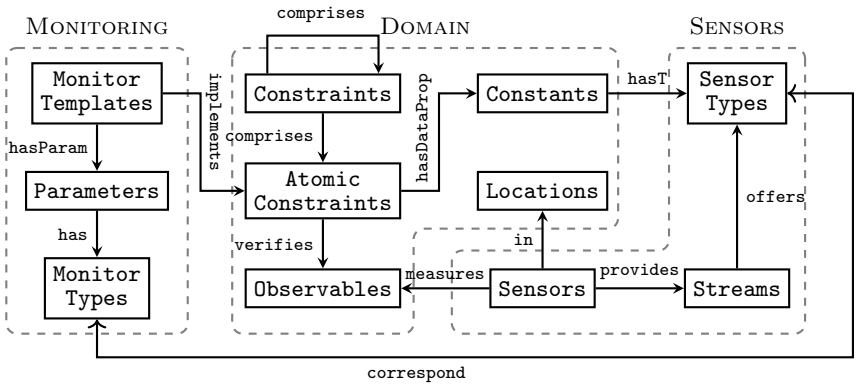


Fig. 6: Knowledge base for the Oslo Fjord Digital Twin.

Monitoring queries $q \in Q(X_{\mathsf{DT}})$ are defined over the following grammar and a variable set X_{DT}. This variable set consists of structured variables of the form `c@l`, where `c` is a constraint and `l` is a location.

$$q ::= x \in X_{\mathsf{DT}} \mid !\, q \mid q \,\&\&\, q \mid q \implies q \mid q \,\|\, q$$

User queries are illustrated in Fig. 7a. The extension of the synthesizer (Defs. 8 and 9) is defined in terms of a function $[q]_{\text{TeSSLa}}$ from queries and constraints to TeSSLa expressions. The knowledge base for the ODT is depicted in Fig. 6. The notion of measurables is slightly more complicated than the general case. The ODT KB introduces *locations* and *observables*, and the measurables are observable-location pairs signifying e.g. "the temperature at location l". This choice aligns with the use case for the system, and leverages location data already in the sensor KB to be used for instrumentation. The construction of the knowledge base required collaboration with a domain expert in marine ecology to describe the ecological niches and the food web for the targeted fish species, such as Atlantic cod (*Gadus morhua*). The knowledge base is implemented as an OWL ontology [1].

Logical consistency (Def. 12) can be checked using a standard OWL reasoner. Data consistency (Def. 13) is checked by a set of graph queries corresponding to the consistency conditions in Fig. 5. Both are static checks that are performed when instantiating the knowledge base. Since both propositional logic and TeSSLa expressions have well-defined semantics we can show semantic soundness of the translation function $[\_]_{\text{TeSSLa}}$ as explained in Sect. 4. In both cases, the semantics is given by interpreting expressions over Boolean streams.

Let $\mathcal{I}_{\text{prop}}$ and $\mathcal{I}_{\text{TeSSLa}}$ be the interpretation functions of propositional logic and TeSSLa-expressions, respectively. The domain of $\mathcal{I}_{\text{prop}}$ is the set $\mathbb{B}$, which is contained in the value domain of TeSSLa by lifting to constant streams. Then for all $q \in \mathcal{Q}(\mathbb{B})$ (i.e. queries without variables) we need to show following property:

$$\mathcal{I}_{\text{prop}}(q) = \mathcal{I}_{\text{TeSSLa}}([q]_{\text{TeSSLa}}).$$

For constraints, the situation is only slightly more complex. We must first define a semantics for our chosen signature $\mathbb{C}$, again into Booleans. The choices are natural: a conjunctive list is true if all of its elements are true, a disjunctive list is true if any of its elements is true, and a negation-list is true if the negations of all of its elements are. Let $\mathcal{I}_{\mathbb{C}}$ denote this interpretation. For all $q \in \mathcal{Q}(\mathbb{B})$:

$$\mathcal{I}_{\mathbb{C}}(q) = \mathcal{I}_{\text{TeSSLa}}(\langle\!|q|\!\rangle_{\text{TeSSLa}}).$$

To evaluate the performance of the synthesizer and the synthesized monitors, we consider a set of monitor queries (shown in Fig. 7a), which are competency questions derived for the integration into the digital twin.

Q1 and **Q2** represent a basic queries: will a particular species thrive in this habitat? **Q1** is unfolded to a conjunction of various preferences for codfish habitats, while **Q2** is atomic. **Q3** shows how variations on a constraint are used to expose a variety of related queries. **Q4** and **Q5** show two different ways in which synthesized monitors can be used to investigate causal relationships. **Q5** is a simple implication, asking if it is currently the case that a species thrives in one location if it thrives in another. In **Q4**, we imagine a constraint that is expensive to monitor, but a "risk"-constraint that is much cheaper. A system use the cheap constraint to monitor risk, only dispatching the expensive constraint to locations where a risk has been detected by the first constraint.

Synthesis of the queries in Fig. 7a is performed in less than 2 s. To evaluate the performance of the resulting TeSSLa monitors, we compiled them with Rust, and used synthetic data streams with varying trace length. The results are shown in Fig. 7 with maximal memory use in Fig. 7b, and per-step time use in Fig. 7c.

6.2 Results and Discussion

We briefly summarize the answers to our research questions:

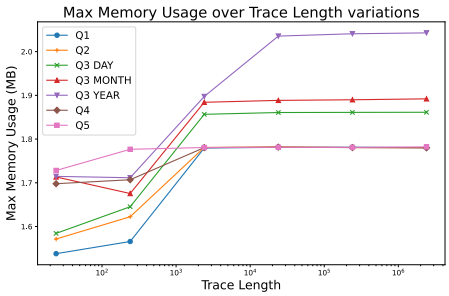
RQ1: we can confirm that it is possible to create knowledge bases and align them with existing domain ontologies and the core knowledge base with some expertise in ontology engineering and with input from domain experts.

RQ2: we can confirm that knowledge graphs are suited for checking logical and data consistency using logical reasoning and graph queries. These findings are in line with previous results on using knowledge graphs in formal methods to provide a data interface with correctness guarantees and modeling methodologies for domain knowledge [30,43]. The proof of language consistency requires manual effort, but this effort can build on existing language semantics.

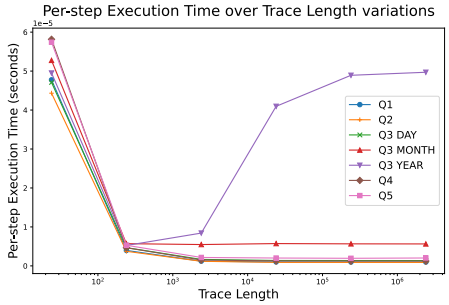
RQ3: in our experiments, synthesizing the runtime monitors is quick ($<$ 2 s) and sufficient for interactive synthesis. In an online setting, there are two important performance constraints on monitors: the time used *per step* must be bounded and the memory usage must be *totally* bounded. The synthesized monitors for the queries in Fig. 7a fulfill both constraints, apart from some start-up noise for traces under $\sim$100 events when dealing with averaging windows. With longer traces, the start-up time is amortized over the entire trace, resulting in a lower per-step time when the results are averaged. Thus, the results for lower

- **Q1:** Is this area a good habitat for a particular species?
  `codPrefersConstraint @ location1`
- **Q2:** Can a particular species find food in this location?
  `codEatsConstraint @ location1`
- **Q3:** Does the temperature in a given period indicate a good habitat for cod?
  `codDayAvgTempConstraint @ location1`
  `codMonthAvgTempConstraint @ location1`
  `codYearAvgTempConstraint @ location1`
- **Q4:** Is there a risk of an algal bloom in a particular location?
  `algaeRiskConstraint @ location1`,
  `algaePrefersConstraint @ location1`
- **Q5:** If one location is a good habitat for a species, is another location also?
  `codPrefersConstraint @ location1 =>`
  `codPrefersConstraint @ location2`

(a) Example user questions and the corresponding monitor queries.



(b) Memory usage.



(c) Execution time.

Fig. 7: Averaged max memory usage (Fig. 7b) and execution time (Fig. 7c) per step normalized over 50 executions for monitors synthesized from the queries in Fig. 7a. The higher per-step time for short traces is caused by start-up costs.

trace lengths are more heavily affected by start-up time execution, as shown in Fig. 7c. **Q3 YEAR** deals with a large time window (8760 h). Its time usage increases as the trace grows to fill this window, but stabilizes (albeit significantly higher than the others) once the window is consistently full.

Modeling assumptions. The main modeling assumption underlying our work is that the knowledge base is sound with respect to the domain knowledge. This assumption is necessary for the soundness of our approach, as it ensures that the synthesized stream specifications are correct with respect to the domain knowledge. In practice, the knowledge base will need to be manually constructed by domain experts, possibly integrating existing ontologies. These experts need to ensure that the resulting knowledge base is sound while following the structure proposed in our work.

Remark that for applications such as a digital twin, the knowledge base may be expected to evolve over time, as new domain knowledge is discovered or new monitoring requirements are introduced. Therefore, it is important to ensure that the knowledge base is regularly updated and maintained while ensuring soundness. Our approach relies on the integration of both domain knowledge and monitoring knowledge, which may require some additional effort to synthesize and maintain the knowledge base.

Observe that our soundness theorems rely on *logical* and *data consistency*. Fortunately, checking these consistency conditions maps to tractable standard tasks on ontologies and knowledge graphs: Logical consistency can be checked using ontology reasoners for OWL [22], while data consistency can be checked using shape constraint languages such as SHACL [46] or ShEx [40].

Scalability and usability. Our approach relies on the synthesis of stream specifications from knowledge bases, and the scalability and usability of our approach may be limited by the size and complexity of the knowledge base. In practice, the size and complexity of the knowledge base might affect the time required to query and reason over the knowledge base, and thereby also the performance of the synthesis process.

The usability of the approach may be limited by the expertise required to construct and maintain the knowledge base itself. However, the underlying structure of the knowledge base allows for modularity and reuse. The knowledge base can be constructed in a modular fashion where different modules can be reused across different domains, as long as the domain knowledge is compatible. This can help reduce the effort required to construct and maintain the knowledge base, as well as improve the scalability and usability of our approach.

7 Related Work

To facilitate the development of temporal logic specifications, patterns [17] and domain-specific languages (DSLs) [24] have been proposed. Some of these DSLs are essentially syntactic sugar that compile into temporal logics, e.g. [4, 6, 8, 45], while others translate directly into executable monitors, such as Tracematches [11] and TeSSLa [13]. Further difficulty arises when deriving correct specifications requires both domain and runtime monitoring expertise. To reduce the effort of repeatedly formalizing domain-specific and monitoring-specific specifications for individual systems, domain expert knowledge can be directly embedded into the runtime verification framework. For example, common specifications in the automotive domain have been collected in a domain-specific TeSSLa module [19], a DSL for flight rules in the space domain [33] translates into TraceContract [5], and an intermediate language for software engineering requirement patterns [26] translates into TeSSLa. A different line of work seeks to produce runtime monitors using LLMs [12] or natural language synthesis from structured requirements [38]. Here, domain knowledge must be included in the query and, lacking correctness guarantees, an expert needs to validate the result.

DynaMo is best positioned between the first two lines of related research, by incorporating expertise from multiple domains. Domain experts can express their knowledge in their own vocabulary, while monitoring experts provide monitoring templates. In contrast to the related work, DynaMo is a general framework that can be instantiated for different domains and different SRVs, allows reuse both with respect to the domain and monitoring knowledge, and in which the formalization enables correctness guarantees.

8 Conclusion

Defining stream specifications for runtime monitoring in complex domains requires both domain expertise and runtime monitoring expertise. This hampers the use of runtime monitoring in domains that depend on agility in specifications, such as self-adaptive systems, IoT-systems and digital twins. This paper addresses this problem by proposing DynaMo, a novel method to synthesize stream specifications from knowledge bases. To generate sound specifications, DynaMo extends (existing) knowledge bases for domain knowledge with monitoring knowledge. Hence, our method bridges the gap between domain and monitoring expertise, while maintaining a separation of concerns between the runtime monitoring framework, the synthesis of stream specifications, and the domain knowledge. We show how to prototypically implement DynaMo and evaluate it on a case study of a digital twin for marine ecosystems. The presented work suggests a novel use of runtime monitors to interface with digital twins, but it also paves the way for research on self-adaptive runtime monitoring in complex domains, in which the monitoring framework itself synthesizes and deploys new monitors triggered by the currently deployed monitors.

References

1. DynaMo repository (2025), https://figshare.com/s/4a8eb447fd5a96495f68?file=60114836
2. Andrews, P.B.: An introduction to mathematical logic and type theory: to truth through proof, Applied Logic Series, vol. 27. Kluwer Academic Publishers, 2 edn. (2002). https://doi.org/10.1007/978-94-015-9934-4
3. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003). https://doi.org/10.1017/CBO9780511711787
4. Barringer, H., Groce, A., Havelund, K., Smith, M.H.: Formal analysis of log files. J. Aerosp. Comput. Inf. Commun. **7**(11), 365–390 (2010). https://doi.org/10.2514/1.49356
5. Barringer, H., Havelund, K.: TraceContract: a Scala DSL for trace analysis. In: Proc. 17th International Symposium on Formal Methods (FM 2011). Lecture Notes in Computer Science, vol. 6664, pp. 57–72. Springer (2011). https://doi.org/10.1007/978-3-642-21437-0_7
6. Bauer, A., Leucker, M., Streit, J.: SALT - structured assertion language for temporal logic. In: Proc. 8th International Conference on Formal Engineering Methods (ICFEM 2006). Lecture Notes in Computer Science, vol. 4260, pp. 757–775. Springer (2006). https://doi.org/10.1007/11901433_41
7. Baumeister, J., Finkbeiner, B., Kohn, F., Löhr, F., Manfredi, G., Schirmer, S., Torens, C.: Monitoring unmanned aircraft: Specification, integration, and lessons-learned. In: Proc. 36th International Conference on Computer Aided Verification (CAV 2024), Part II. Lecture Notes in Computer Science, vol. 14682, pp. 207–218. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_10

8. Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., Rodeh, Y.: The temporal logic Sugar. In: Proc. 13th International Conference on Computer Aided Verification (CAV 2001). Lecture Notes in Computer Science, vol. 2102, pp. 363–367. Springer (2001). https://doi.org/10.1007/3-540-44585-4_33
9. Bezerra, C., Freitas, F., Santana, F.: Evaluating ontologies with competency questions. In: Proc. IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology. pp. 284–285. IEEE Computer Society (2013). https://doi.org/10.1109/WI-IAT.2013.199
10. Biron, P.V., Peterson, D., Malhotra, A., Thompson, H., Gao, S., Sperberg-McQueen, M.: W3C XML schema definition language (XSD) 1.1 part 2: Datatypes. W3C recommendation, W3C (Apr 2012), https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/
11. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. J. Log. Comput. **20**(3), 707–723 (2010). https://doi.org/10.1093/LOGCOM/EXN077
12. Cohen, I., Havelund, K., Peled, D., Goldberg, Y.: The power of reframing: Using LLMs in synthesizing RV monitors. In: Proc. 25th International Conference on Runtime Verification (RV 2025). Lecture Notes in Computer Science, vol. 16087, pp. 202–212. Springer (2025). https://doi.org/10.1007/978-3-032-05435-7_12
13. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: Temporal stream-based specification language. In: Proc. 21st Brazilian Symposium on Formal Methods: Foundations and Applications (SBMF 2018). Lecture Notes in Computer Science, vol. 11254, pp. 144–162. Springer (2018). https://doi.org/10.1007/978-3-030-03044-5_10
14. Cox, S., Perego, A., Browning, D., Beltran, A.G., Winstanley, P., Albertoni, R.: Data catalog vocabulary (DCAT) - version 3. W3C recommendation, W3C (Aug 2024), https://w2.syronex.com/jmr/w3c-biblio
15. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: Proc. 12th International Symposium on Temporal Representation and Reasoning (TIME 2005). pp. 166–174. IEEE Computer Society (2005). https://doi.org/10.1109/TIME.2005.26
16. Dimou, A., Sande, M.V., Colpaert, P., Verborgh, R., Mannens, E., de Walle, R.V.: RML: A generic language for integrated RDF mappings of heterogeneous data. In: Proc. Workshop on Linked Data on the Web, co-located with the 23rd International World Wide Web Conference (WWW 2014). CEUR Workshop Proceedings, vol. 1184. CEUR-WS.org (2014), https://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf
17. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. International Conference on Software Engineering (ICSE'99). pp. 411–420. ACM (1999). https://doi.org/10.1145/302405.302672
18. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) Proc. 16th International Conference on Runtime Verification (RV 2016). Lecture Notes in Computer Science, vol. 10012, pp. 152–168. Springer (2016). https://doi.org/10.1007/978-3-319-46982-9_10
19. Friese, M.J., Kallwies, H., Leucker, M., Sachenbacher, M., Streichhahn, H., Thoma, D.: Runtime verification of AUTOSAR timing extensions. In: Proc. 30th International Conference on Real-Time Networks and Systems (RTNS 2022). pp. 173–183. ACM (2022). https://doi.org/10.1145/3534879.3534898

20. Gangemi, A., Presutti, V.: Ontology design patterns. In: Handbook on Ontologies, pp. 221–243. International Handbooks on Information Systems, Springer (2009). https://doi.org/10.1007/978-3-540-92673-3_10
21. García-Castro, R., Lefrançois, M., Poveda-Villalón, M., Daniele, L.: The ETSI SAREF ontology for smart applications: a long path of development and evolution. In: Energy Smart Appliances, chap. 7, pp. 183–215. John Wiley & Sons, Ltd (2023). https://doi.org/10.1002/9781119899457.ch7
22. Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P., Sattler, U.: OWL 2: The next step for OWL. Journal of Web Semantics **6**(4), 309–322 (2008). https://doi.org/10.1016/j.websem.2008.05.001
23. Havelund, K., Goldberg, A.: Verify your runs. In: Proc. First Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2005). Lecture Notes in Computer Science, vol. 4171, pp. 374–383. Springer (2005). https://doi.org/10.1007/978-3-540-69149-5_40
24. Havelund, K., Omer, M., Peled, D.: DSLs for runtime verification. In: Proc. 25th International Conference on Runtime Verification (RV 2025). Lecture Notes in Computer Science, vol. 16087, pp. 22–43. Springer (2025). https://doi.org/10.1007/978-3-032-05435-7_2
25. Havelund, K., Reger, G., Rosu, G.: Runtime verification past experiences and future projections. In: Computing and Software Science - State of the Art and Perspectives, Lecture Notes in Computer Science, vol. 10000, pp. 532–562. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_25
26. Hipler, R., Kallwies, H., Leucker, M., van Dommele, K.G., Wien, J.: A practical approach to runtime verification. In: Proc. 25th International Conference on Runtime Verification (RV 2025). Lecture Notes in Computer Science, vol. 16087, pp. 377–396. Springer (2025). https://doi.org/10.1007/978-3-032-05435-7_21
27. Hitzler, P.: A review of the semantic web field. Commun. ACM **64**(2), 76–83 (2021). https://doi.org/10.1145/3397512
28. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., Ngomo, A.N., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J.F., Staab, S., Zimmermann, A.: Knowledge graphs. ACM Comput. Surv. **54**(4), 71:1–71:37 (2022). https://doi.org/10.1145/3447772
29. Huisman, M., Monti, R.E.: On the industrial application of critical software verification with VerCors. In: Proc. 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA 2020), Part III. Lecture Notes in Computer Science, vol. 12478, pp. 273–292. Springer (2020). https://doi.org/10.1007/978-3-030-61467-6_18
30. Kamburjan, E., Bencomo, N., Johnsen, E.B., Tapia Tarifa, S.L.: Declarative lifecycle management for self-adaptive systems. Softw. Syst. Model. **25**(3), 787–814 (2026). https://doi.org/10.1007/S10270-026-01358-W
31. Kamburjan, E., Slaughter, L.A., Johnsen, E.B., Pferscher, A., Weihl, L.: Towards self-adaptive data management in digital twins for biodiversity monitoring. In: EDTConf. IEEE (2025). https://doi.org/10.1109/MODELS-C68889.2025.00044
32. Keet, C.M., Khan, Z.C.: On the roles of competency questions in ontology engineering. In: Proc. 24th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2024). Lecture Notes in Computer Science, vol. 15370, pp. 123–132. Springer (2024). https://doi.org/10.1007/978-3-031-77792-9_8
33. Kürklü, E., Havelund, K.: A flight rule checker for the LADEE lunar spacecraft. In: Proc. 17th International Colloquium on Theoretical Aspects of Computing

(ICTAC 2020). Lecture Notes in Computer Science, vol. 12545, pp. 3–20. Springer (2020). https://doi.org/10.1007/978-3-030-64276-1_1
34. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams. In: Proc. 33rd Symposium on Applied Computing (SAC 2018). pp. 1925–1933. ACM (2018). https://doi.org/10.1145/3167132.3167338
35. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebraic Methods Program. **78**(5), 293–303 (2009). https://doi.org/10.1016/J.JLAP.2008.08.004
36. Libkin, L.: How to define certain answers. In: Yang, Q., Wooldridge, M.J. (eds.) Proc. 24th International Joint Conference on Artificial Intelligence (IJCAI 2015). pp. 4282–4288. AAAI Press (2015), http://ijcai.org/Abstract/15/609
37. Lipski Jr., W.: On semantic issues connected with incomplete information databases. ACM Trans. Database Syst. **4**(3), 262–296 (1979). https://doi.org/10.1145/320083.320088
38. Perez, I., Mavridou, A., Pressburger, T., Goodloe, A., Giannakopoulou, D.: Automated translation of natural language requirements to runtime monitors. In: Proc. 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022), Part I. Lecture Notes in Computer Science, vol. 13243, pp. 387–395. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_21
39. Poveda-Villalón, M., Fernández-Izquierdo, A., Fernández-López, M., García-Castro, R.: LOT: an industrial oriented ontology engineering framework. Eng. Appl. Artif. Intell. **111**, 104755 (2022). https://doi.org/10.1016/J.ENGAPPAI.2022.104755
40. Prud'hommeaux, E., Gayo, J.E.L., Solbrig, H.R.: Shape expressions: an RDF validation and transformation language. In: SEMANTiCS. pp. 32–40. ACM (2014). https://doi.org/10.1145/2660517.2660523
41. Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstic, S., Lourenço, J.M., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019). https://doi.org/10.1007/S10703-019-00337-W
42. Shvaiko, P., Euzenat, J.: Ontology matching: State of the art and future challenges. IEEE Trans. Knowl. Data Eng. **25**(1), 158–176 (2013). https://doi.org/10.1109/TKDE.2011.253
43. Sieve, R., Kamburjan, E., Damiani, F., Johnsen, E.B.: Declarative dynamic object reclassification. In: Proc. 39th European Conference on Object-Oriented Programming (ECOOP 2025). LIPIcs, vol. 333, pp. 29:1–29:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025). https://doi.org/10.4230/LIPICS.ECOOP.2025.29
44. Skjæveland, M.G., Karlsen, L.H.: The Reasonable Ontology Templates Framework. Transactions on Graph Data and Knowledge **2**(2), 5:1–5:54 (2024). https://doi.org/10.4230/TGDK.2.2.5
45. Vardi, M.Y.: From Church and Prior to PSL. In: 25 Years of Model Checking - History, Achievements, Perspectives. Lecture Notes in Computer Science, vol. 5000, pp. 150–171. Springer (2008). https://doi.org/10.1007/978-3-540-69850-0_10
46. World Wide Web Consortium (W3C): Shapes constraint language (shacl) (2017), https://www.w3.org/TR/shacl/, w3C Recommendation, Accessed: 2026-08-10

47. Xu, X., Mao, Z., Su, J., Lin, X., Basin, D.A., Sun, J., Wang, J.: Quantitative runtime monitoring of Ethereum transaction attacks. In: Proc. ACM on Web Conference (WWW 2025). pp. 4146–4159. ACM (2025). https://doi.org/10.1145/3696410.3714682
48. Zapridou, E., Bartocci, E., Katsaros, P.: Runtime verification of autonomous driving systems in CARLA. In: Proc. 20th International Conference on Runtime Verification (RV 2020). Lecture Notes in Computer Science, vol. 12399, pp. 172–183. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_9