

An extensible approach to implicit incremental model analyses

Georg Hinkel¹ · Robert Heinrich² · Ralf Reussner²

Abstract

As systems evolve, analysis results based on models of the system must be updated, in many cases as fast as possible. Since usually only small parts of the model change, large parts of the analysis' intermediate results could be reused in an incremental fashion. Manually invalidating these intermediate results at the right places in the analysis is a non-trivial and error-prone task that conceals the codes intention. A possible solution for this problem is implicit incrementality, i.e., an incremental algorithm is derived from the batch specification, aiming for an increased performance without the cost of degraded maintainability. Current approaches are either specialized to a subset of analyses or require explicit state management. In this paper, we propose an approach to implicit incremental model analysis capable of integrating custom dynamic algorithms. For this, we formalize incremental derivation using category theory, gaining type-safety and correctness properties. We implement an extensible implicit incremental computation system and validate its applicability by integrating incremental queries. We evaluate the performance using a micro-benchmark and a community benchmark where the integration of explicit query incrementalization was multiple orders of magnitude faster than rerunning the analysis after every change.

Keywords Incremental computation · Model-driven · Monads

1 Introduction

In many engineering disciplines, abstract models of a system are created in order to reason on properties of the modeled system by analyzing the model. Nowadays, many of these systems are supported by software that runs the analyses automatically based on in-memory representations of the models. As the systems evolve, the models are changed and the analyses may have to be recomputed. These analyses include simple validations, but also more complex model transformations or simulations.

For some analyses, large benefits can be obtained from incremental execution. An early example is digital circuit

simulation where incremental simulation yields orders of magnitudes in performance [1,2] by introducing buffers to save some intermediate results.

Saving such intermediate results for future requests on changed input models and their invalidation is called incremental derivation or incrementalization. The goal of this process is that ideally, only those parts of an analysis that are affected by a change in the underlying models have to be recomputed. This may offer an increased performance as the savings in terms of reused intermediate results can be larger than efforts to invalidate those results affected by a change. This is important since in many application areas, the response time to get updated analysis results for a given model change is critical. Examples include the area of self-adaptive systems where it is important to reconfigure the system as fast as possible before it crashes or breaks service level agreements [3]. Hence, this response time is the most common measurement for evaluating approaches in the area of self-adaptive systems [4].

In other application areas like graphical user interfaces with the Model-View-Viewmodel pattern [5,6], it is especially important to get change notifications for analysis results in order to update the view accordingly to a changed model underneath.

Communicated by Dr. Daniel Varro.

✉ Georg Hinkel
georg.hinkel@gmail.com

Robert Heinrich
heinrich@kit.edu

Ralf Reussner
reussner@kit.edu

¹ Wiesbaden, Germany

² Karlsruhe Institute of Technology, Am Fasanengarten 5,
76131 Karlsruhe, Germany

Manual incrementalization is a non-trivial and error-prone task, as it is very easy to forget cases in which intermediate results need to be invalidated. Furthermore, the management of intermediate results may conceal the analysis code and degrade understandability. This makes it harder to proofread the code, thus leading to undetected bugs in the analysis and hence wrong analysis results implying wrong conclusions on the real system. Besides correctness, understandability is crucially important. Currently, understanding existing code makes up almost half of software maintenance costs [7]. Maintenance in turn is the main driver for overall software project costs [8].

Furthermore, in some cases even little changes have a dramatic effect; thus, keeping prior intermediate results does not yield any benefits. Therefore, manual incrementalization may turn out to not give any benefits, despite the efforts put into them. Whether or not an analysis benefits from incremental execution is often hard to foresee. In addition, a batch (i.e., non-incremental) version of the analysis is often nevertheless desirable, if the analysis is also used in cases where model changes can be ignored. Here, manual incrementalization leads to duplicated code, e.g., one version with buffers optimized for incremental execution and one without these buffers, saving memory and event management.

A promising approach to tackle this problem is implicit incrementality, also referred to as implicit self-adjusting computation. In this approach, the system decides which intermediate results should be saved and manages their invalidation, typically by tracking its dependencies. This process is transparent to the analysis developer since no changes to the analysis are required. Such systems exist either for general-purpose languages capable of expressing any analysis [9] or for specific classes of analyses such as incremental queries [10,11], incremental pattern matching [12] or even incremental model transformations [13,14]. These specialized incremental approaches limit their applicability to a given class of analyses and use abstractions common to these analyses to make incremental execution more efficient.

General-purpose incrementalization systems such as the type-directed self-adjusting computation system [9] or Adaption [15,16] typically operate on a Turing-complete calculus (such as the λ -calculus) and are thus able to incrementalize any analysis. To do that, a dynamic dependency graph (DDG) is deduced from the execution of the running analysis. This graph is then used to decide when parts of the analysis should be reevaluated for given inputs. Thus, the incrementalized version of an analysis is tightly coupled to its batch specification. Rather on a technical level, the existing approaches are not well suited for model analysis, as they mostly rely on immutable data structures known from functional programming. Models, however, are usually mutable and several concepts such as resetting references once a

model element is deleted or bidirectional references are not well suited for immutable data structures.

The exact conformance to the batch specification can lead to inefficient results. Consider the minimalistic example of incremental average calculation for a list of model elements and a predicate that should be collected. The typical batch implementation is to iterate through the list and keep a running sum and element count. If the predicate changes for any model element in the list, this results in different running sums from that index on. Incrementalization based only on the batch specification has to take this into account and therefore recalculate the average calculation starting from this index in the collection. The more efficient implementation would be to only increment the running sum and element count, ignoring where exactly the predicate was changed.

Given that many functions such as average computation are very common and used across many different analyses, it is reasonable to invest additional effort to optimize these functions once for incremental execution by supplying a dedicated dynamic algorithm. However, current approaches do not provide a technique to do this, at least not for higher-order functions.

Special-purpose incrementalization systems circumvent this problem of low-level incrementalization because the domain-specific operations they operate on, such as joining partial matches of a graph pattern [12], are already on a high abstraction level, but cannot describe all kinds of analysis. This can be problematic in evolution scenarios, if analyses fall out of the selected scope. Furthermore, the applicability of these tools is limited to a certain class of analyses, for example, graph patterns.

The goal of the research presented in this paper is therefore to overcome aforementioned limitations and provide an approach to integrate developer-supplied dynamic algorithms into implicit incremental computation. We present a formalization of incrementalization as a functor. This allows us to define the requirements for a dynamic algorithm that shall be integrated, and prove correctness under these assumptions. In particular, this includes a formal definition of the semantics that a dynamic algorithm has to implement such that the correctness of the incrementalized analysis can be guaranteed by construction when exchanging the batch algorithm by a dynamic counterpart.

In the aforementioned example of average calculation, our approach gives developers a way to specify an incremental average computation by implementing a class that keeps a running sum and element count. This class should track the selected predicate for each model element in the underlying collection and adjust the running sum if any changes in the predicate arise, independently of the index of the respective element in the collection. The explicit incrementalization has to be specified once by developers of the average function and

can be used many times without additional efforts, whenever an analysis requires an average computation.

Such an optimization is cost-effective since higher-order operators are typically used in a broad range of scenarios. We therefore see two roles: The framework developer that implements generic analysis frameworks that aid the general creation of model analyses, and the analysis developer that applies these frameworks to his particular domain of interest to implement a concrete information need.

To validate our results, we have implemented a query framework¹ that supports incrementalization of model analyses using our approach through the C# query syntax. The presented approach is based on the .NET Modeling Framework (NMF, [17,18]). We evaluated the performance improvements in a micro-benchmark where we were able to see that manually incrementalized methods are multiple orders of magnitude faster than both instruction-level incrementalization or batch reevaluation. We further evaluate the performance against the incremental graph pattern matching tool EMF- INCQUERY [11] in a community benchmark [19]. In this benchmark, our approach is faster than the specialized approach EMF- INCQUERY for many model sizes.

Our implementation lets developers specify analyses in the mainstream language C# so that typical problems of domain-specific languages, such as tool support availability [20,21] and language adoption [22], are mitigated. The resulting incrementalized analysis supports online update notifications and offers better response times from model changes to updated analyses at the price of higher memory consumption. Our approach is applicable to general-purpose analyses, but can also incorporate efficiency based on analysis abstractions. The implicit approach allows us to derive these benefits without the cost of degraded understandability, conciseness or maintainability as the developer does not have to change the analysis code. In summary, we make the following contributions:

- C1 We present a novel formalization of incremental computation using functors from category theory (Sect. 3) based on a new formalization of mutable models (“Appendix A.2”). This formalization adopts the idea of Carlsson [23] to represent incrementalization using category theory, but for mutable models.
- C2 We propose an approach to integrate analysis frameworks into incremental computation systems in order to use the framework abstractions (Sect. 4).
- C3 We show how the formalization of the incrementalization concept can be used to implement an implicit incremen-

talization system with support for higher-order functions (Sect. 5).

- C4 We apply our approach to a query framework (Sect. 6) and evaluate it in a micro-benchmark and in a community benchmark for incremental pattern matching (cf. Sect. 7).

Furthermore, our approach has already been used to power incremental model synchronization [24–27] using our incremental model analysis approach as a building block without an explanation of the underlying theory as presented in this paper.

Before we describe these contributions, we motivate our approach with an illustrative example (Sect. 2). Section 3 introduces our concept how we formalize incremental computation and the theorem that we can prove based on this formalization. This section is written in an informal style. Formal details and proofs can be found in “Appendix A”. The sections that follow are written in a rather technical style. Sect. 4 explains the general idea of integrating dynamic algorithms and discusses the advantages and subtleties using an example from graph algorithms. Section 5 presents the implementation of our extensible incrementalization system NMF Expressions. Section 6 presents the integration of query support as a demonstration for the extensibility. Section 7 evaluates the incrementalization system and its query extension using a micro-benchmark of computing averages and the TTC 2015 Train Benchmark. Finally, we discuss related work in Sect. 8 and conclude the paper in Sect. 9. We discuss future work in Sect. 10.

2 Running example

Throughout the paper, we use a synthetic example analysis to both explain and evaluate our approach. The example is taken from the TTC Train Benchmark [19]. Though only a synthetic benchmark, this example case demonstrates both practical use cases and also many of the problems typical for incremental computation.

One of the tasks in this benchmark is to select the switches along routes in a railway net that are set incorrectly according to signal positions. The railway network is described in a model conforming to a railway metamodel created by Szárnyas et al. [19]. An illustration of an instance model for a railway network excerpt is depicted in Fig. 1.

The railway network essentially consists of many segments, switches, semaphores and routes. Each route starts and ends at a semaphore and is defined by a list of switch positions which define where a train following this route should go. An excerpt of the metamodel is depicted in Fig. 2.

One wants to make sure that if the entry semaphore shows the signal *GO*, all switches along the route should be set

¹ Here, we mean collection queries inspired by SQL as we see them in a variety of programming languages nowadays. This is opposed to queries as in the query-or-command pattern where queries are side effect-free methods.

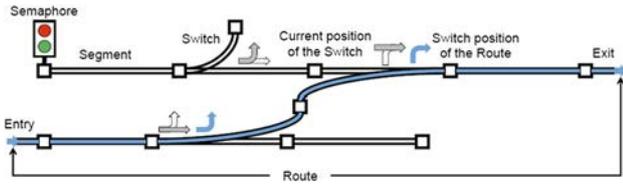


Fig. 1 A visualization of the railway network instance model as used in the TTC 2015 Train Benchmark case [19]

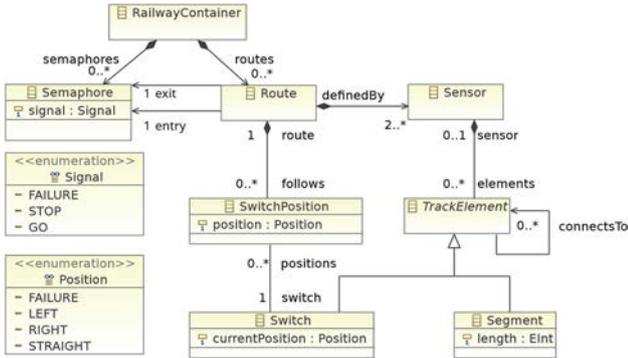


Fig. 2 An excerpt of the railway metamodel used in the Train Benchmark [19]

```

1 var faultyPositions = from route in
  routes
2   where route.Entry != null && route
   .Entry.Signal == Signal.GO
3   from swP in route.Follows
4   where swP.Switch.CurrentPosition
   != swP.Position
5   select swP;

```

Listing 1 Query to find inaccurate switch positions in a collection of routes routes

according to the route description. The benchmark iteratively finds and fixes some (10 or 10%) violations of this and some other validation constraints.

A possible solution to this analysis is the NMF solution which can be found in Listing 1. According to the peer-reviewed process at the TTC 2015, this solution was the most understandable, even for developers not familiar to the C# language.

Line 1 takes as input a collection of all routes in the network. Line 2 selects those routes that have an entry semaphore set and that entry semaphore shows a *GO* signal. Line 3 selects the switch positions along those routes that define which switches in the network have to be set to what position. Line 4 restricts the set of switch positions in the result set of the query to those where the position of the corresponding switch does not match the required position. Lastly, Line 5 specifies that the result set of the query should only contain the switch position elements.

While this solution is very hard to beat in terms of understandability and conciseness, using standard C#, the entire model has to be reevaluated whenever a model element changes. Furthermore, one has no information when the analysis should be reevaluated and a new result has to be compared with the last one in order to understand which switch positions are wrong that were not wrong before.

As a consequence, as soon as performance becomes an issue, developers may start introducing cache objects, e.g., to save the routes with an entry semaphore set to the signal *GO* and dynamically registering hooks when the position of switch positions changes. However, this is a laborious and error-prone procedure as one may easily forget some cases when to update these caches. For example, one may easily forget to remove the hooks when a *SwitchPosition* element is removed from one of the routes with *GO* signal. While this maintains correctness, it slowly decreases the performance over time and is therefore hard to detect.²

Presumably the most dramatic consequence of such an analysis inflated by caches is that domain experts likely have no longer a chance to proofread the code. In contrast, the code in Listing 1 is likely to be understood by railway experts, meaning they could identify possible flaws in the understanding of what this analysis should do.

Therefore, the goal of (live) implicit incrementality is to enable the system to execute the analysis from Listing 1 incrementally. That is, the system automatically registers event handlers to propagate model changes and issues a notification if a change to the model caused the result of the analysis to change. However, to achieve good results, it is necessary to provide an explicit incrementalization of commonly used functions such as the query operators *from*, *where* and *select* used in Listing 1. Enabling incrementalization systems to incorporate such dynamic algorithms is made possible through C2.

For example, if the query in Listing 1 is running and any model manipulation operation changes the position of a switch, then the incremental system would automatically recheck all switch positions pointing to this switch on routes with entry signal *GO*, whether the position declared by the route matches the actual position of the switch. For this to work efficiently, the system caches the routes that have an entry signal set to *GO* as well as the switch positions on this route. These caches are automatically maintained, transparently to the developer.

Further, if the incrementalized analysis resembled the batch specification on a low abstraction level and the entry signal of a given route changes to *GO*, then all of the subsequent routes would have to be inspected again, because the batch analysis would do this as well. Approaches such as

² Some approaches exist that may automatically detect such performance problems by automatically conducting experiments [28].

Nominal Adapton [16] try to mitigate this by reducing the effort to recheck routes to a lookup, but a system understanding the *where*-operator on a high abstraction level does not need to recheck any of the other routes because the wrongly set switches along these routes have not changed. Our approach provides a mechanism to specify such semantics once for higher-order operators such as the *where*-operator.

As a very important set of methods used in a range of analyses, we have identified the Standard Query Operators (SQO³) which the compiler uses to map the query syntax, though the approach is also applicable for other higher-order methods as well.

NMF supports incremental model analyses, including a dedicated support for incremental queries based on the SQOs, through the subproject *NMF Expressions*, which is explained in depth in Sects. 5 and 6.

```
1 using NMF.Expressions.Linq;
```

Listing 2 A namespace import to NMF.Expressions.Linq

To accomplish incrementalization, all the analysis developer has to do is to insert a namespace import at the top of the file such as in Listing 2 and to make sure that the model classes implement two interfaces to propagate elementary changes. The latter is not a hard restriction since these interfaces are also used by user interface technologies on the .NET platform. The model code generated by NMF for a given metamodel implements these interfaces as well, such that no additional effort is necessary, if a metamodel is present.

The effect of the `using`-statement is that the C# compiler will use a different set of methods to bind the SQO methods used in Listing 1. These methods implement our approach so that their return value supports an event to notify clients when new elements were added to the result. In case of the `faultyPositions` query, this happens, for example, when new switch positions arise along a route where the entry semaphore shows *GO* but their switch is not set accordingly, for example because either the routes entry signal or the switch position has just changed. It may also happen when the position of a switch changes, because there is some route with a green entry signal that follows this switch, because the switch element is mutable, i.e., the position of the switch changes even though the switch is still the same.

3 Incrementalization as a functor

This section roughly presents our approach to formalize incremental computation (C1). For brevity, details such

³ <http://msdn.microsoft.com/en-us/library/bb394939.aspx>; SQO is a set of language-independent standard APIs for queries, specifically defined for the .NET platform.

as formal constructions and proofs have been moved to “Appendix” of this paper.

The goal of a formalization for incremental computation systems given an analysis morphism $f : M \rightarrow R$ is some object of a type $\mathcal{I}(R)$. This object will represent the running live analysis. From this object, we would like to query the current analysis result and apply any model changes or get notified when the value has changed.

Functors from category theory are an adequate tool to formalize such kind of relationships between methods (in category theory terms: morphisms). Therefore, our goal is to reify incrementalization as a functor that maps an analysis $f : M \rightarrow R$ to its incrementalization $\mathcal{I}(M) \rightarrow \mathcal{I}(R)$.

To obtain the current value and apply changes, we suggest the mappings

$value : \mathcal{I}(R) \rightarrow R$ and $apply : \Delta\Omega \rightarrow (\mathcal{I}(R) \rightarrow \mathcal{I}(R))$.

In this situation, the *value* function is meant to return the current value of an incremental value instance of $\mathcal{I}(R)$ while *apply* applies a change in the global state to the incremental value. The idea is that this application could be used to propagate changes to the analysis result. The type $\mathcal{I}(R)$ is dependent on R to maintain type-safety while the system \mathcal{I} is independent of the analysis result type. In this setup, *value* and *apply* can be formalized as natural transformations,⁴ in order to reflect their independence of the result type.

As a trivial example, consider the check whether a semaphore is set to *GO* in the running example. Let $Signal : Semaphore \rightarrow Signal$ be the property getter returning the current signal of a semaphore. Further we have the morphism $\neq : Object \times Object \rightarrow bool$ and the constant value *GO* can be extended to a constant morphism $GO : T \rightarrow Signal$ which returns the signal *GO* regardless of the state that is provided as a parameter. Thus, we can formulate the expression as

$$isGo : Semaphore \rightarrow bool,$$

$$(s, \omega) \mapsto \neq (Signal(s, \omega), GO(\omega)).$$

In implementations, such a representation can be easily retrieved from a typed abstract syntax tree.

With the help of mutable type categories (cf. “Appendix A.2”), we can formalize incremental analyses using functors, i.e., we can formalize \mathcal{I} from above as a functor from the type system \mathcal{C}_Ω where we only consider side effect-free methods for incrementalization. The functor \mathcal{I} then maps each type A in \mathcal{C}_Ω to some $\mathcal{I}(A)$ in \mathcal{C}_Ω for which we demand the existence of a natural transformation $value : \mathcal{I} \rightarrow Id_{\mathcal{C}_\Omega}$ and a mapping *apply* that maps state changes in $\Delta\Omega$ to natural endotransformations of \mathcal{I} . We can then apply \mathcal{I} to our

⁴ *apply* is actually a family of natural transformations indexed by $\Delta\Omega$.

analysis f (since the latter is assumed side effect free and therefore in \mathcal{C}_Ω) and yield a function $\mathcal{I}(f) : \mathcal{I}(M) \rightarrow \mathcal{I}(R)$. We call this function with a constant reference of the input model and obtain an incremental result object in $\mathcal{I}(R)$. This object is then used to automatically update analysis results from a changed model underneath.

Using the functor \mathcal{I} , we can apply it to our small subexpression `isGo` to retrieve

$$\mathcal{I}(\text{isGo}) : \mathcal{I}(\text{Semaphore}) \rightarrow \mathcal{I}(\text{bool}).$$

The associativity of the functor guarantees us that we can assemble $\mathcal{I}(\text{isGo})$ from the functor applied to the components of `isGo`, i.e., its abstract syntax tree. For an implementation, this means that only an implementation for the elements of the abstract syntax tree has to be provided. These implementations can then be stacked together to realize the implementation of the generic functor.

When a model change $\Delta\omega \in \Delta\Omega$ occurs in state ω , we apply it to the incremental result $r \in \mathcal{I}(R)$ using the apply function by evaluating $\text{apply}(\Delta\omega)_R(r)$. Whenever the current value of the analysis is needed, it can be obtained using *value*.

Moreover, the general approach of using functors to change the way how a given function is executed is independent of the exact structure of the type $\mathcal{I}(A)$ for a given type A . This provides several degrees of freedom for implementations.

Finally, we arrive at the following definitions:

Definition 1 (Incremental Computation System) Let Ω be a set of global states. Let \mathcal{C} be a mutable type category (cf. ‘‘Appendix A.2’’). Then an incremental computation system $\mathcal{I} : \mathcal{C}_\Omega \rightarrow \mathcal{C}_\Omega$ for \mathcal{C} is a functor for which the natural transformation $\text{value} : \mathcal{I} \rightarrow \text{Id}_{\mathcal{C}_\Omega}$ and the function $\text{apply} : \Delta\Omega \rightarrow (\mathcal{I} \rightarrow \mathcal{I})$ exist where *apply* targets the natural endotransformations of \mathcal{I} . We further demand a (non-natural) transformation $\eta : \text{Id}_{\mathcal{C}_\Omega} \rightarrow \mathcal{I}$ with stateless components such that

$$\text{value} \circ \eta = \text{Id}_{\text{Id}_{\mathcal{C}_\Omega}}$$

$$\text{apply}(\text{Id}_\Omega) = \text{Id}_\mathcal{I}.$$

The first equation demands that the composition of *value* and η is the identity functor on \mathcal{C}_Ω . In particular, it is natural (even though η is not natural). In other words, lifting a value to a constant and then asking that constant for its current value should obtain the original value.

The second equation means that *apply* changes neither the given incremental value nor the global state if the identity on the state space is passed in, i.e., no changes have occurred. This is similar to hippocraticness requirements of synchronization systems.

Last, we demand that applying a state change to constants does not have an effect, i.e., we have that for each $\Delta\omega \in \Delta\Omega$ that

$$\text{apply}(\Delta\omega) \circ \eta = \eta \circ \Delta\omega.$$

Here, we used the inclusion defined in Definition 15.

For correctness, we want incremental values giving us the same analysis results as we would obtain through batch mode execution. This is formalized by the below definition.

Definition 2 (Correctness of Incremental Computation Systems) An incremental computation system \mathcal{I} on the category \mathcal{C} is correct if for every A and B in \mathcal{C} , every side effect-free morphism $f : A \rightarrow B$ in \mathcal{C}_Ω and every state change $\Delta\omega \in \Delta\Omega$ the following holds:

$$\text{value}_B \circ \mathcal{I}(f) \circ \eta_A = f \quad (\text{Initialization})$$

and

$$\text{value}_B \circ \text{apply}(\Delta\omega)_B \circ \mathcal{I}(f) \circ \eta_A = f \circ \Delta\omega_A. \quad (\text{Updates})$$

as morphisms $A \rightarrow B$. This corresponds to the following commutative diagram for (Initialization):

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \eta_A \downarrow & & \uparrow \text{value}_B \\ \mathcal{I}(A) & \xrightarrow{\mathcal{I}(f)} & \mathcal{I}(B) \end{array}$$

The equation (Updates) corresponds to the following commutative diagram:

$$\begin{array}{ccccc} A & \xrightarrow{\Delta\omega_A} & A & \xrightarrow{f} & B \\ \eta_A \downarrow & & & & \uparrow \text{value}_B \\ \mathcal{I}(A) & \xrightarrow{\mathcal{I}(f)} & \mathcal{I}(B) & \xrightarrow{\text{apply}(\Delta\omega)_B} & \mathcal{I}(B) \end{array}$$

In the last diagram, the mapping $\text{apply}_B(-, \Delta\omega)$ is the mapping $\mathcal{I}(B) \rightarrow \mathcal{I}(B)$, $b \mapsto \text{apply}_B(b, \Delta\omega)$.

This means, if we create an incremental value for a given analysis and immediately query the current value, we get the same as if we just executed the original analysis (Initialization). Before we do that, we can apply a state change $\Delta\omega \in \Delta\Omega$ to the incremental value and then it should give us the same value as if we were obtaining the analysis result value again from scratch (Updates).

The key observation here is that while on the right hand of (Updates), the analysis function f is only used after the

state change $\Delta\omega$ is applied, the left hand of the equation first evaluates $\mathcal{I}(f)$ before applying the change using apply_B . As a consequence, we already know the analysis f when we apply $\Delta\omega$ and can use abstractions of f to update caches.

Theorem 1 *Let \mathcal{I} be an incremental evaluation system for the MTC \mathcal{C} . Then \mathcal{I} is correct.*

Proof The proof is given in appendix, see “Appendix A.4”. \square

Remark 1 Theorem 1 essentially shows that the correctness of an incrementalization system is a consequence of the naturality of the *value* and *apply* transformations. These naturalities can be checked for each morphism separately and thus enable to deduce the correctness of an entire incrementalization system from the correct incrementalization of elementary morphisms. As a reason, the commutative diagrams that are required for the naturality of a transformation can be easily stacked together as long as the functor conforms to the law that $\mathcal{I}(f \circ g) = \mathcal{I}(f) \circ \mathcal{I}(g)$. Thus, if a transformation is natural for morphisms f and g , it automatically is natural for $f \circ g$.

Remark 2 In general, *apply* is allowed to change the global state. This can be useful in case that change propagation should imply subsequent model changes, for example, to recompute derived features.

4 Integrating dynamic algorithms into incremental analyses

This section describes how arbitrary frameworks or libraries can be tuned for implicit incremental computation systems (C2). Many analyses are based on recurring problems with dedicated algorithmic solutions for the incremental (dynamic) and non-incremental case, often based on graph theory. In the literature, the APIs for both kinds of algorithm are different: The API for the dynamic algorithm usually extends the API for the non-incremental case by operations that propagate input changes. For our approach, this is problematic because we assume a model analysis to be strictly separated from the model manipulation—the functor \mathcal{I} is only defined for side effect-free methods. In particular, we do not want to make the model manipulation aware that there is an incremental analysis going on. Rather, the analysis has to adapt to the changed model automatically. Therefore, the goal of this section is to describe how algorithms need to be reified for implicit incrementalization in order for the incrementalization system to choose a different algorithm in the incremental case than what is executed in the batch case.

For this, we first explain why different algorithms are necessary in the incremental case and then present the approach how such problems must be reified for incrementalization.

4.1 Choice of algorithms

As an example for graph algorithms beyond queries, we have chosen connectivity analysis to explain our approach. This means, we analyze whether two nodes in a graph are connected, i.e., whether there is a path between them. We chose this type of analysis not because of its relevance in practice but because it demonstrates the problems of choosing algorithms well, meanwhile in our running example, the choice is rather obvious.

In batch mode, one would typically use a *Union-Find* data structure that is created in $\Theta(n + m\alpha(n))$ time [29] and answers connectivity queries in $O(\log n)$ time where n is the number of vertices, m is the number of edges and α is the inverse Ackermann function [30]. This amounts to $\Theta(n + m\alpha(n))$ when we answer at most $O(\log n)$ connectivity queries. As Tarjan showed, this solution has optimal asymptotic complexity [29].

The Union-Find data structure essentially adds a parent-pointer to each vertex pointing to a representative of its strongly connected component. These pointers are followed until an element is found which references itself. Then, two vertices are in the same cluster iff their pointers ultimately point to the same element. The data structure is created by iterating through all edges and making sure that vertices connected by an edge are always in the same cluster.

The Union-Find data structure does not support decremental updates, i.e., when edges are removed from the graph, the entire data structure has to be rebuilt. However, there is also a fully dynamic connectivity algorithm by Holm et al. [31]: He suggests to create and maintain a data structure of dynamic spanning forests in the graph, thus answering connectivity queries in amortized logarithmic time $O(\log n)$ while requiring amortized $O(\log^2 n)$ time for updating the data structure when edges are inserted or deleted. This yields a total time of $O(\log^2 n)$ to update analysis results on model changes when we are only answering $O(\log n)$ connectivity queries.⁵

The key observation here is that the incremental algorithm in this case, maintaining a dynamic spanning forest, is entirely different from the batch mode approach of using a Union-Find data structure. While Tarjan’s Union-Find data structure efficiently answers connectivity queries, the dynamic spanning forest by Holm can be updated even if edges are deleted from the graph.

However, although Holm’s dynamic spanning forest algorithm is known for more than a decade, it can be doubted that many analysis developers are aware of it or even can implement it. For a developer of an analysis, it is more common to use an implementation of connectivity analysis provided by a library, without a deeper understanding of the algo-

⁵ A performance comparison of different algorithms was done by Cattaneo et al. [32].

rithm that is used behind the scenes. The rationale behind our approach is that the developer of that library probably knows the dynamic algorithms available, but requires a way to implement that algorithm in a way such that an incrementalization system is able to access this implementation.

Another important observation is that the choice of algorithms also depends on the usage context: Although Holm’s dynamic spanning forests can greatly improve the amortized asymptotic update performance in case connectivity is only interesting for few pairs of nodes, it eventually gets worse than recreating the Union-Find data structure from scratch after every change. The choice which of these algorithms are faster for a given practical model size may therefore be difficult to answer.

4.2 Reification of the problem for incrementalization

Incrementalization approaches that work on an instruction level do not see the algorithmic problem and therefore are not aware that there may be an elegant solution in an incremental setting which is entirely different to the best solution in the batch scenario. Further, most graph algorithms are specified in imperative code that modifies some internal state in loops, where some loop invariant ensures the correctness. As the state has substantial influence on the control flow, an incrementalization based on the batch specification has to keep track of the potentially huge state space or reevaluate the algorithm from the earliest state that diverged, as implemented in Traceable Data Types [33].

However, developers of a framework for graph connectivity likely know such an alternative solution as they are aware of the methods semantics and could specify a dynamic version. The analysis developer can reuse the connectivity analysis as a building block and the incrementalization automatically decides whether to run the connectivity analysis using Tarjans *Union-Find* data structure or Holms dynamic spanning forest, depending on whether the analysis is executed in batch mode or incrementally.

The basic idea is to enable developers to provide a custom implementation of the functor application (cf. Sect. 3) of framework functions. An explicit functor application is only required once for each generic analysis method such as connectivity analysis while it may be used in a multitude of analyses. An examples are the SQO methods that we manually incrementalized for our motivational example in Sect. 2.

The advantage of our formalization of incrementality as a functor is that the correctness of the whole analysis follows from the requirement that functors respect functional composition, i.e., for morphisms $f : A \rightarrow B$, $g : B \rightarrow C$ we have that $\mathcal{I}(f \circ g) = \mathcal{I}(f) \circ \mathcal{I}(g)$.

In terms of programming languages, this means that a function f from $A_1 \times \dots \times A_n \rightarrow B$ with n parameters must

be mapped to a function $\mathcal{I}(f) : \mathcal{I}(A_1) \times \dots \times \mathcal{I}(A_n) \rightarrow \mathcal{I}(B)$. If any of the A_i is a function type, then the incrementalization of this function may utilize the functor \mathcal{I} on arguments passed for this parameter. This allows to provide explicit incrementalizations for higher-order functions.

For example, consider again Listing 1. The query in this listing is translated into calls to higher-order functions such as `where` and `select` operators (SQOs). The incrementalization of these operators can now take the incrementalization of the predicates used by these functions into account, regardless of how these functions look like, and keep a reference on the incremental results that are return values of the incrementalized predicates.

Calls to these higher-order functions need to be mapped to calls of the incremental derivatives, i.e., functions to which the incremental computation system \mathcal{I} has been applied. An easy specification method is possible in languages that keep metadata such as Java or C#. The metadata of a function can then contain a reference to the incremental derivative, e.g., through `.NET` attributes or Java annotations.

While this approach is a straightforward outcome of our formalization (C1), it has a strong impact on the API design of analysis frameworks. In algorithmics, fully dynamic algorithms such as the connectivity algorithm from above are often designed with an API that mixes the functional specification of the algorithm (in the example a function returning whether two vertices are connected) and an API to adjust the data structure to updated input (in the example methods that insert or remove edges from the graph). As a consequence of our approach, the latter is forbidden but must be implemented as part of the functor application.

In the example of connectivity analysis, we can reduce the API of a generic connectivity analysis implemented in a single class to the two elements below.

- `Connectivity<T>` ($T^* \text{ vertices}, T \rightarrow T^* \text{ edges}$) : `Connectivity<T>` creates a new data structure to decide whether two elements of type T are connected where the underlying graph is given by a set of vertices and for each vertex the incident edges. Here, T^* denotes the Kleene closure, i.e., a collection of type T .
- `AreConnected(T a, T b)` : `bool` as an instance method of the resulting data structure determines whether the vertices `a` and `b` are connected.

In batch mode, the method `Connectivity` creates a *Union-Find* data structure as proposed by Tarjan. On this data structure, the method `AreConnected` checks for two instances of the domain, whether the parent pointers are pointing to the same element.

In the incrementalized version, the result of `Connectivity` is an incremental value of a connectivity object created

using Holms dynamic spanning trees, inheriting from the same abstract base class. These methods get as an input an incremental value for the vertices in the graph and an incremental value for the method describing the outgoing edges. This object will react on changes in the vertices appropriately by adding or removing edges in the dynamic forest. If, for example, the value for the parameter `edges` changes entirely, it may also return a new `Connectivity` object, meaning that the present dynamic forest is discarded.

The method `AreConnected` of the incremental dynamic spanning tree implementation compares the root nodes for both involved trees and looks whether they match. Furthermore, it hooks an event handler to react on changes to the dynamic forest and reruns the check afterward. The resulting incremental Boolean value represents whether this has any effect on the connection between nodes `a` and `b`.

This can be seen as a separation of concerns in the otherwise query-and-command like interface of fully dynamic algorithms. In this version, the functionality is exposed in a purely functional manner whereas the state management is hidden from the developer when the analysis is run in incremental mode.

5 An extensible implicit incremental computation system in .NET

This section presents NMF EXPRESSIONS, an extensible implementation of an incremental computation system (C3). The system is available as open source⁶ as a subproject of NMF [17] which supports model-driven engineering on the .NET platform. NMF Expressions is an implementation for online incrementality, i.e., we assume that the analysis is run in a long-running process and gets notified for each elementary model change.

The basic idea of NMF Expressions is to implement incremental expression evaluation by creating a dynamic dependency graph (DDG) where each executed instruction is reflected by a node in the DDG. DDG nodes represent incremental values and are annotated with their current value (*value* transformation).

As usually many instructions are necessary to compute an analysis, DDGs may become very large and may consume enormous amounts of memory. This makes the integration of manually incrementalized functions important to avoid that graph traversal outweighs the savings in terms of incremental computation.

In the remainder of this section, we first introduce the overall concept in Sect. 5.1, discuss the correctness of the implementation in Sect. 5.2, the incrementalization of higher-order methods in Sect. 5.3 and the extensibility in Sect. 5.4.

⁶ <http://github.com/NMFCode/NMF>.

5.1 Overview

To implement an incremental computation system, one of the first decisions to make is how to design the API of the incrementalization, i.e., how to implement the functor.

For the mapping of types, our implementation uses a generic interface `INotifyValue` for the mapping of types to decouple the monad⁷ as much as possible from concrete implementations. Furthermore, interfaces in .NET offer support for covariance whereas in general, .NET uses a hard implementation of generics in .NET.⁸ This means, if a class `A` derives from `B`, then an `INotifyValue<A>` object can be assigned to a variable of type `INotifyValue`.

The mapping of functions, i.e., computing $\mathcal{I}(f)$ for a given function f , is done at runtime in our implementation. This works because .NET languages such as C# have a feature to obtain a function, in particular, a lambda expression, as a pre-compiled abstract syntax tree (expression tree). Hence, a function f is represented by such an expression tree that is generated by the compiler. This allows us to lift this function at runtime while keeping the tool support intact. The price is a slight overhead to the non-incremental execution of such a function as the expression has to be compiled before it can be used in non-incremental mode. In the incremental case, our implementation relies on conversions from the pre-compiled lambda expressions to a new set of types `ObservingFunc` with a set of generic type arguments for parameter types and a generic type parameter for the return type, similar to the .NET built-in delegate `Func` types. These `ObservingFunc` types represent a function f together with its incrementalization $\mathcal{I}(f)$ and $\mathcal{I}(f) \circ \eta$. That is, one can either call the function with the regular parameters and get the regular result (for batch purposes), call the function with incremental values for each parameter and get an incremental result or call the function with the regular parameters and get an incremental result where all arguments are lifted to constants.

Next, we need to define how state changes ($\Delta\Omega$) should be mapped to the type system in order to provide an implementation of the update mechanism (*apply* transformation). In our implementation, *apply* is supported through an explicit method at the individual DDG nodes that receives changes from the DDG nodes it depends upon and outputs the value changes in the current node. DDG nodes in turn are realizations of incremental values. That is, whenever a state change occurs, the change propagates through the DDG. If the value

⁷ The unit transformation of an incrementalization system is not natural, therefore incrementalization is not a monad in the sense of category theory, only a functor. However, this construct is often still called a monad in programming.

⁸ That is, generic type arguments matter for the identity of a type, unlike for instance in Java.

of a node changes, this change is propagated to all of the successor nodes; otherwise, change propagation is stopped.

In a modeling environment, the state changes are model changes that can be recorded using standard notification APIs. NMF reuses the notification API that is common in the .NET platform, available through the interfaces `INotifyPropertyChanged` and `INotifyCollectionChanged`.⁹ Because the implementation only uses these two interfaces, it can also be used with model classes that are not generated from a metamodel but written directly.¹⁰

In the formalization, we considered DDG nodes immutable as components of \mathcal{I} . In the implementation, we essentially reuse the old object and fire an event when a DDG node should be considered as different. The advantage is that we can describe the actual changes in the notification, for instance to note the exact value change, i.e., the old and the new value. The same is done for collections where the notification (in the form of an event) also carries information which objects have been added or removed (cf. Sect. 6).

In addition to the *value* and *apply* transformations, the DDG nodes also carry a reference counter in order to track whether the incremental value that they represent is currently needed. If it is not, the DDG nodes are in a deactivated state and do not listen to model changes.

The unit transformation η is straightforward to implement as extension methods. The unit transformation η converts a value to a constant, i.e., a DDG node that always returns the provided constant as its value and never changes upon calls of *apply*. The functor itself is not as easy. In our implementation, we decompose methods into their abstract syntax trees and incrementalize every element of it separately, making use of the law that $\mathcal{I}(f \circ g) = \mathcal{I}(f) \circ \mathcal{I}(g)$.

For this to work, we require a decomposition of the model analysis into instructions. We obtain this decomposition at run time through the .NET Expression API in an expression tree of the `System.Linq.Expressions` namespace.¹¹

The resulting DDG essentially contains a node for each executed instruction, including the type of instruction as well as the data passed in. It is therefore much larger than comparable DDGs created by self-adjusting computation [36] that uses explicit incrementalization primitives to make the nodes as large as possible. However, it has the advantage that we have a direct representation of a method call which makes it

easier to exchange such nodes with an explicit incrementalization for the given method.

If any node in the expression tree changes its value, this change is propagated up to the root of the tree that represents the value of the whole tree. Along this way, the propagation is stopped as soon as the value for a subexpression does not change.

For example, whether the entry semaphore of a route shows *GO* does not change if the entry semaphore of the route has a failure while showing *STOP* (depicted in Fig. 3). The member access node to the entry semaphore does not change because the identity of the semaphore is still the same. However, the signal property of that semaphore changed. This change raises an event, fetched by the member access node and further propagated through the DDG. The node for the binary operator `==` receives the change notification and checks whether it should be updated. However, the signal still does not show *GO* and thus the change is no longer propagated.

If a node in a DDG is referenced by two other DDG nodes—which means that an intermediate result is used more than once—this causes problems as some DDG nodes may be reevaluated too often.¹² However, for many analyses, this problem does not occur. Therefore, NMF Expressions provides multiple implementations of reevaluation strategies called execution engines that support change transactions or even parallel change propagation. However, we consider these different execution engines outside the scope of this paper.

5.2 Incrementalization at instruction level

We implemented an incrementalization for each instruction type, each represented in its own class. If a change affects an incremental value, we do not exchange the instance of the DDG node but issue a change notification such that dependent nodes treat the incremental value as new. The expression tree is then converted using a visitor pattern. For each of the instruction types, their incrementalization has to respect the naturality of *value* and *apply* transformations.

The naturality of *apply* means that

1. the creation of the DDG can be done before or after a given change is done to the model without affecting the DDG after the change and
2. the change notification is issued whenever the represented value changes.

⁹ The interface `INotifyCollectionChanged` is only used in the extension for incremental queries in Sect. 6.

¹⁰ The support for these two interfaces can even be generated automatically using aspect-oriented programming [34].

¹¹ C# has an option to instruct the compiler to generate a model of the code instead of compiling the code to Intermediate Language (IL) code. The usage of this language feature to build internal DSLs has been discussed first by Martin Fowler [35, p. 455] under the term *Parse Tree Manipulation*.

¹² For example, consider the *repeated-diamonds-shape* problem illustrated on <http://rystsov.info/warp9/pages/competitors/diamond/diamond.html>.

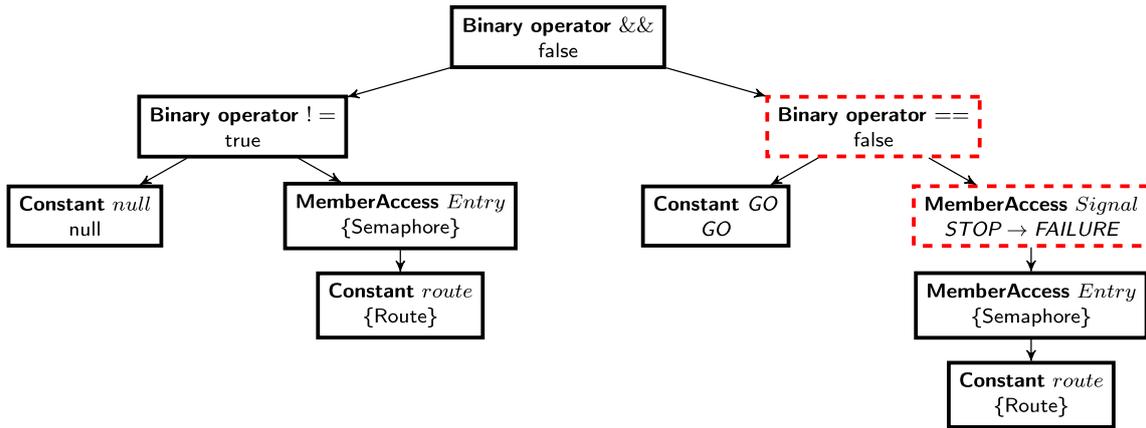


Fig. 3 The DDG for the predicate `route.Entry!=null&&route.Entry.Signal==Signal.GO` and nodes that must be reevaluated when changing the signal to `FAILURE` in red and dashed (color figure online)

The former statement is true for all nodes of our implementation, as the implementation is entirely sequential and therefore the creation of a DDG node cannot interfere with the change propagation. As soon as a change happens, all DDG nodes that are affected by this change adapt themselves to the change. The latter statement and the naturality of the *value* transformation have to be discussed for each instruction type individually. This means that at any given global state, the *value* transformation of a DDG node must match the instruction applied to the *value* of the input DDG node and if this result has changed since the last model manipulation, a change notification must have been issued. This change notification may contain detailed information on the change that may help to propagate it. An implementation for the most common types of instructions is described below.

Constants Constants never change. Thus, the node does not save successor nodes as a notification that the value has changed does not occur. The *value* transformation is also a constant, which is clearly natural.

Member access A member access potentially changes either if the target model element for the member access changes or any change in the target element's properties is recorded through the notification API.

Unary expressions The considered unary expressions are type casts, conversions, unary plus and minus of numbers, logical negation and bitwise inverse. These operators only change their value when their inputs change.

Binary expressions The value of a binary expression potentially changes if either of the operand's values changes. An exception to this rule is the logical shorthand operators. In case of the conditional shorthand `&&` operator, the right operand must not be evaluated if the left operand evaluates to `false`, as it might throw an exception. Thus, the right operand must be activated or deactivated,

depending on the value of the left operand. The same applies for conditionals.

Conditional expressions Conditional expressions keep a DDG for the test expression, the true expression and the false expression. Depending on the current value of the test DDG, the DDG for the true or false expression are dynamically activated and deactivated. The value of the conditional expression only changes if the value for the attached sub-DDG root node changes.

Method calls, constructors In case we have an abstract syntax tree of the method available such as for Lambda expressions, we recursively deduce a dependency graph template from it. In all other cases, we assume that a method return value only changes if either of its arguments changes. This assumption is reasonable for immutable types, particularly for platform functions like string length or the sinus function to which we do not have access. In all other cases, we require that the developer provides an explicit incrementalization of the method.

Lambda expressions Nested lambda expressions are slightly problematic. Because the function types of the .NET platform are fixed, using a custom function type loses the inherited compiler support.¹³ Therefore, the approach of NMF is to perform a lazy incrementalization of lambda expressions. That is, the lambda expression is only incrementalized when actually needed.¹⁴ If this is the case, the body expression of the function is recursively transformed into the monad as well.

Invocations An invocation of a function expression is very similar to a method call with the exception that no proxy

¹³ One may circumvent this problem by extending the compiler. Using technologies such as Roslyn, this seems possible, even though one then also has to extend the IDE support.

¹⁴ In case of nested predicates, the inner lambda expressions need to remain lambda expressions to respect type system laws.

is required as the incrementalization system knows the body of the function expression. The result is that the corresponding DDG template (cf. Sect. 5.3) is filled with the dependency graph nodes obtained from the arguments.

Dynamic dependency graphs consume a lot of memory and are the main reason why incremental computation has a large memory overhead. Therefore, approaches like the implicit self-adjusting computation by Chen et al. [9] argue that constant operations that do not change their value should not go into the functor since they unnecessarily increase the size of the DDG. To solve this problem, their approach generates methods for each combination of an incremental value¹⁵ and constant value. To circumvent this problem, we introduced a constant propagation, i.e., we do create nodes in the dynamic dependency graph if a value is constant (i.e., there is no change notification provided for it) but reduce operations made on constants to constant values.

Converting the abstract syntax trees at runtime yields the decision whether or not we apply the monad. If so, we can apply the monad and obtain an incremental evaluation. If we do not apply the monad, we can use the .NET built-in expression compiler and get a batch mode version of the analysis with low overhead: Because the types of all expressions are already known, it is straight forward and thus fast to compile an expression tree to intermediate language code.¹⁶

5.3 Incrementalization of higher-order functions

Many model analyses such as the detection of wrongly set switches in the running example include the usage of higher-order functions, i.e., functions that take functions as arguments. This has two consequences: First, the transition from a function to a DDG happens while the system is running, potentially as a reaction to some model changes. Therefore, the parsing process of functions to DDGs need to be moved upfront. Second, we need to handle cases where the argument function changes while the system is running.

To solve the first problem, we use templates of DDGs. The DDG is created for the body of the function, using placeholders whenever an argument is accessed. Upon creation, the entire DDG for a function is in a disconnected state. If arguments are passed to the system, the DDG template is copied, replacing the argument placeholders with the provided DDG nodes. If all parameters are satisfied, the DDG is connected. Otherwise, the copied DDG stays disconnected and realizes the exponential mate, also known as the curried version of the original method.

¹⁵ Called modifiable reference in [9].

¹⁶ However, the generated method should be stored in order to avoid repeated JIT-compilation.

DDG templates are also used for conditional and shorthand binary expressions. For example, the right side of a shorthand `&&` operator must not be evaluated if the left side already evaluates to `false`. Therefore, in that case we deactivate the subgraph. For the predicate `route.Entry!=null &&route.Entry.Signal== Signal.GO` of the running example, this is depicted in Fig. 4.

However, as an incremental analysis is usually meant to run continuously, it is very important that the algorithm is elastic in its memory consumption. This means, the memory of DDG nodes is released once they are no longer needed.

In our implementation, each DDG node has a separate counter to determine whether it should be connected or disconnected, because a DDG node generally does not know where it is used. If this reference counter is incremented to 1, the node automatically connects which means that it increments the reference counter for all of its prerequisite nodes and attaches to the model notification API if necessary. Conversely, if the reference counter is decremented to 0, the DDG node disconnects from the model and decrements the reference counters of prerequisite nodes. However, the implementation still holds a strong reference to the DDG nodes such that they are not collected by the garbage collector. This is because otherwise it would not be possible to connect to the model again.

With parts of the DDG being able to be removed while the incremental analysis is running, we can solve the problem how to properly represent higher-order functions. A higher-order function compatible with our approach is a function that takes in a pre-compiled lambda expression as a parameter (as above, in order to keep the tool support). The incrementalization of this function would then consume an incremental value of pre-compiled lambda expressions and turns the current value into a DDG template. If the value of the function parameter ever changes (which rarely happens in all scenarios we have come across so far), the DDG fragment created for the template is discarded and replaced with a fragment created from the new DDG template. In order to keep track of the borders of the DDG fragment, wrapper DDG nodes are inserted, if necessary.¹⁷

5.4 Extensibility

To keep the DDGs in a more reasonable size and improve efficiency or to support also non-pure methods, we allow developers to provide a custom incrementalization of a given function. If such a manual incrementalization is provided, the function is no longer seen as a composition of instructions but rather treated as a primitive. The moment a manual incrementalization is specified, the restrictions for the method no

¹⁷ The wrapper is not necessary for instance if the function parameter is a constant.

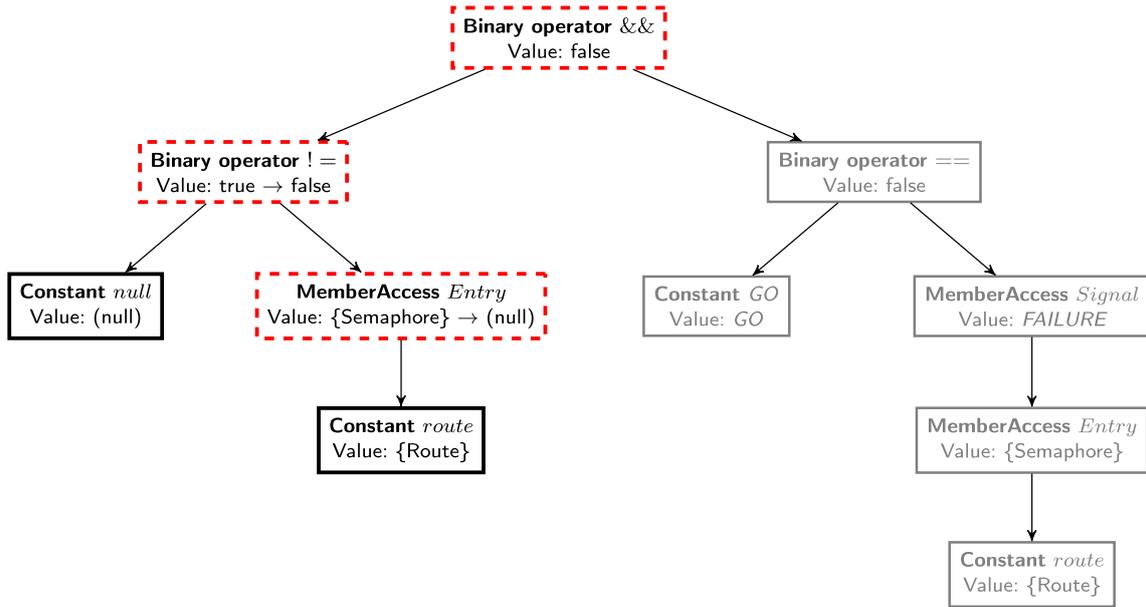


Fig. 4 The DDG for the predicate `route.Entry!=null&&route.Entry.Signal==Signal.GO` and nodes that are disconnected if the entry semaphore is changed to `null`

longer apply. This means, the method could also be implemented in an imperative manner, such as, for example, graph algorithms usually are.

We allow users different variants of specifying a proxy. For a function $f : A \rightarrow B$, the user may either provide a function $\mathcal{I}(f) : \mathcal{I}(A) \rightarrow \mathcal{I}(B)$ or a function $f' : A \rightarrow \mathcal{I}(B)$. In case of the latter, we essentially reevaluate f' whenever the current value of the arguments change and wrap the resulting incremental value of B . The rationale behind this decision is that for many functions—in particular higher-order functions—such as the *Average* aggregate,¹⁸ it is much easier to specify f' and it does not cost much performance since the internal memory has to be reset entirely when the original arguments change (which often does not even occur). Furthermore, especially in the presence of collection operators, the argument identities of most collection operators often do not change.

For the actual specification of the manual incrementalization, we use an annotation called `ObservableProxy`. This annotation specifies a type and a method name, identifying a method that realizes the given extension point.

A problematic situation arises, if the method is recursive. To create a DDG template that contains a recursive method, we have to avoid recursively calling the proxy method. Here we make use of the fact that the DDG template for the method is only needed when the method is actually called, i.e., when the DDG node is activated. In particular, we use a proxy node that only copies the DDG template for the required method

¹⁸ Here, we consider an overload of `Average` that takes as input a function what the value of an element shall be.

as soon as the node is connected to the model. As this proxy node means additional memory, we require the user to specify whether the proxy method is recursive.

A common problem with incremental proxies is that we have no option but to trust the developer that the provided proxy is correct in the sense that it fulfills the naturality of *value* and *apply* for the annotated method. However, because the requirements of the proxy are clear in relation to the annotated method, a check of these naturalities could be a goal of future code verification activities.

6 Incremental queries as an example extension

This section presents an implementation of our concepts for incremental queries. As queries are popular, this implementation is also part of NMF but separated in its own assembly, proving that the incremental computation system is independent from the query implementation.

Query operators are a good candidate for incrementalization because the dynamic algorithms for most operators are easier than for other algorithms such as connectivity analysis algorithms. Meanwhile, the added value in terms of asymptotic improvements is larger: Maintaining a running sum and element count to compute an average for instance reduces the recalculation of an average from linear to constant effort, despite its simplicity. Unfortunately, for other algorithms, it is often neither as easy nor are the advantages as clear, as discussed in detail in Sect. 4.1.

Because the incrementalization system is independent from the incremental query support, we claim that this proves the extensibility of our incrementalization system. Furthermore, because most query operators are higher-order functions, the incremental query framework also demonstrates the support for higher-order functions.

According to the formalization, *apply* should always create a different object, if the current value of an incremental value has changed. For collections, one usually wants to avoid this scenario as it implies to copy a lot of memory. Furthermore, one usually wants to abstract from the kind of collection (NMF typically uses an array list or a hash set or a combination of both, depending on order and uniqueness constraints of a reference, plus customizations for opposite and containment references).

Dynamic algorithms usually require collection changes at a high level. Therefore, we essentially use the same idea as for DDG nodes, we treat collections as immutable but require them to fire an event whenever their contents change and fetch these events in manual incrementalizations of collection operators. Because the generic incrementalization system is unaware of collections, appropriate dynamic algorithms are even *necessary* because the general incrementalization system is not aware of collection changes and would otherwise lead to wrong results as collections in NMF are mutable. The collection change event is hidden behind an extended collection interface. This interface yields a high-level change representation of collections, similar to the proposal of Cai et al. [37], making abstractions from the concrete collection implementation. This change representation enables us to abstract from the index of a changed element in a collection or even the collection implementation.

Queries can be seen as an extension of collections into a monad [38]. Thus, we only refine this monad to represent changes, i.e., combine them with the `INotifyCollectionChanged` interface commonly used in the .NET platform for collection changes. That is, instead of the usual `IEnumerable` interface, we created a new `INotifyEnumerable` interface for incremental computation and the `IEnumerableExpression` that allows users to switch between batch mode and incremental mode. `IEnumerableExpression` behaves like the `IEnumerable` monad but allows to switch to the `INotifyEnumerable` monad through a method call.

The extension of collections to a monad is supported on the .NET platform through the SQO methods (cf. Sect. 2). For each of these methods, we have defined a manual functor implementation that enables to use them incrementally. The `INotifyEnumerable` monad is already fixed to incremental execution. That is, any operation on this monad operates directly on a DDG node representing a collection. However, in applications, the SQO opera-

tions are also often helpful in a context where incremental computation is not required. Therefore, we added the `IEnumerableExpression` monad that models an analysis. This model can then be executed in batch mode (the default) or turned into a DDG upon request.

We implemented the following extension methods that are part of the SQO both for the `INotifyEnumerable` monad and for the `IEnumerableExpression` monad: *All*, *Any*, *Average*, *Cast*, *Concat*, *Contains*, *Count*, *Distinct*, *Except*, *FirstOrDefault*, *GroupBy*, *GroupJoin*, *Intersect*, *IsProperSubsetOf*, *IsProperSupersetOf*, *IsSubsetOf*, *IsSupersetOf*, *Join*, *Max*, *Min*, *OfType*, *OrderBy*, *OrderByDescending*, *Select*, *SelectMany*, *SetEquals*, *Sum*, *ThenBy*, *ThenByDescending*, *Union* and *Where*. The semantics of these extension methods match their definitions from the SQO which are reflected by their names. We implemented the overloads that do not consider element indices that are thus not available on either of our monads. If element indices are considered, an insertion of an element often results in too many changes for incremental execution to be beneficial. In particular, adding or removing an element from a collection of n elements in the average leads to $\frac{n}{2}$ index changes, meanwhile if indices are not considered, only the removed element needs to be adjusted. Furthermore, these overloads are not considered in C# for the query syntax and are thus rather rarely used.

We demonstrate the implementation of the manual incrementalization for the *Average* function. The average of a collection (for example, of integers) changes not only when the identity of its argument collection changes, but also when the contents of this collection change as we are using mutable collections. Therefore, *Average* plays the role of f from Sect. 5.4.

For each overload of the *Average* function, a proxy method is provided that implements a manually incrementalized version. Since our formalization in category theory maintains types, these proxy methods can be type checked and used without any conversion efforts. At the same time, we know that the incremental analysis is still correct, assuming that the custom incremental derivation for each method is correct.

The implementation of the *Average* method including the annotation that specifies the manual incrementalization is depicted in Listing 3. The proxy method is specified using a type and the name of a public method. If the original method is static, the proxy method must also be static. The proxy method must have either the same input parameters or monad instances of them (in the example of Listing 3, `INotifyValue<IEnumerableExpression>`), corresponding to the cases that the proxy is provided as f' or $\mathcal{I}(f)$ from Sect. 5.4. The proxy method itself switches the input collection to a version that enables collection updates and returns a custom DDG node implementation.

```

1 [ObservableProxy(typeof(Proxies), "
  CreateIntAverage")]
2 public static double Average(this
  IEnumerableExpression<int> list)
3 {
4   ...
5 }
6 private static class Proxies {
7   public static INotifyValue<double>
8   CreateIntAverage(
  IEnumerableExpression<int>
  list) {
9     return new ObservableIntAverage(
10    list.AsNotifiable())
11   };
12 }
13 }

```

Listing 3 Definition of the *Average* method and specification of the manual incrementalization

A simplified version¹⁹ of the implementation of this DDG node is depicted in Listing 4. In particular, the DDG node memorizes the current sum and element count. When the input collection resets, sum and element count are reset. Otherwise, these fields are updated according to the changes of the underlying collection.

The DDG node implementation may assume that only change notifications from DDG nodes listed as dependencies will be passed to this node. In the example, the average node may safely assume that all change notifications come from the collection DDG node *source*. Notably, because the notification result is hidden behind an interface, extensions may easily implement their own change representations such as the query implementation uses the change representation of mutable collections.

The explicit incrementalizations of higher-order methods such as the *Select* or *Where* operators internally manage DDGs for any element in the underlying collection. If an element in the collection is added, a new DDG is created to obtain an incremental value for the predicate the operator is using. If the element is removed from the collection, the DDG is no longer needed and removed from the node.²⁰ If the result of one of the element DDG change, the corresponding change in the operator is deduced and then propagated.

¹⁹ The real implementation is refactored to share code with similar aggregation functions and catches error cases. A few names have been simplified.

²⁰ The implementations are aware of cardinalities larger than 1 so that effectively, the DDG is removed if the cardinality of the removed element is 0.

```

1 internal class ObservableIntAverage :
  NotifyValue<double>
2 {
3   private int sum;
4   private int count;
5   private double value;
6   private INotifyEnumerable<int> source
  ;
7
8   public ObservableIntAverage
  (INotifyEnumerable<int> source) {
9     this.source = source;
10  }
11
12  public override IEnumerable<
  INotifiable>
13  Dependencies {
14    get { yield return source; }
15  }
16
17  public override INotificationResult
  Notify(ICollection<INotificationResult>
  changes) {
18    var old = value;
19    foreach (ICollectionChange change
20    in changes) {
21      if (change.IsReset)
22      { sum = 0; count = 0; }
23      foreach (int added in change.
24      Added)
25      { sum+=added; count++; }
26      foreach (int removed in change.
27      Removed)
28      { sum-=removed; count--; }
29    }
30    value = (double)sum / count;
31    if (old != value) {
32      return new
  ValueChangedNotificationResult
33      <double>(this, old,
  value);
34    } else {
35      return
  UnchangedNotificationResult.
  Instance;
36    }
37  }
38  public override double Value { get {
  return value; } }
39 }

```

Listing 4 The implementation of the *Average* DDG node

7 Validation and evaluation

In this section, we perform a validation and evaluation of our approach. Section 7.1 explains the goals and strategy behind our evaluation. Section 7.2 presents a micro-benchmark to validate that manual incrementalizations improve the efficiency of an incrementalized algorithm. Section 7.3 evaluates

our performance in a realistic scenario and compares the results with other, both incremental and non-incremental tools. Section 7.4 discusses threats to validity. Section 7.5 gives a summary of the validation.

7.1 Goals and strategy

In the scope of this paper, we have the following validation and evaluation goals:

1. We want to validate that the extension mechanism introduced in this paper has a better performance (measured in response times to changes as well as memory consumption and size of models that can be processed) than a comparable version without the extension mechanism.
2. We want to validate that model analyses incrementalized using our approach achieve better response times than comparable non-incremental solutions.
3. We want to evaluate the performance of model analyses incrementalized using our approach in comparison with other incremental tools.

To maximize the validity of our results, we aim to use community benchmarks for which a wide range of solutions using other tools exist. For brevity, we have only included one of such benchmarks, the TTC 2015 version of the Train Benchmark [19], in this paper. For further evaluations, the interested reader is referred to the Ph.D. thesis of the first author [39] or the NMF solution [40] of the TTC 2018 Social Media benchmark [41].

All of these benchmarks require to deal with collections, for which NMF uses its manually incrementalized query framework (cf. Sect. 6). An implementation of these queries without the abstraction of collections (in order to avoid manual incrementalizations) would make these queries much more difficult to understand than their batch equivalents and therefore has not been implemented. To compare these manually incrementalized functions with the default incrementalization using recursive functions, we added a new micro-benchmark that compares the computation of an average using the optimized manual implementation or a instruction-level incrementalized recursive algorithm.

7.2 An incremental average micro-benchmark

In this section, we analyze a small micro-benchmark in order to validate the efficiency gains by using manual incrementalizations of higher-order functions. The code for this benchmark is publicly available at <https://github.com/NMFCode/ExtensibilityBenchmark>.

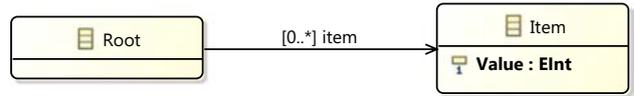


Fig. 5 The trivial metamodel used for the average micro-benchmark

7.2.1 Benchmark setup

To measure the performance of computing the average of a collection, we created a trivial metamodel that we use as basis. The metamodel only consists of two classes: a class `Root` that serves as container and a class `Item` that has an attribute `Value` containing the integers for which we want to compute the average value. The metamodel is depicted in Fig. 5.

Based on this metamodel, the benchmark consists of the following steps:

1. Generate a model of size $n + 1$ consisting of one `Root` element and n instances of `Item` with values for the `Value` attribute drawn by random between 0 and 100.
2. Perform 40 model changes. Each model change is either to add a new item with a random value or to modify the value of a randomly selected existing item. After each change, we recalculate the average of all elements. The first 20 model changes are discarded, for the second 20 model changes we measure the time required to recalculate the average. The random numbers are created outside the time measurements.
3. The process is repeated 50 times for each size n to eliminate the factors of just-in-time compilation and randomly occurring garbage collections.

We make the benchmark deterministic by seeding the random number generators and check the correctness of all implementations by comparing the sums of all averages per size and algorithm and making sure they are all the same.

We created a small benchmark framework where an implementation essentially consists of two methods: `Initialize` and `ComputeAverage`. The former method is executed at the start of the benchmark, the latter after every model change. We compared five different implementations:

- A batch implementation that uses the regular, non-incremental query operator `Average`
- An incremental implementation that uses a manually incrementalized version of the `Average` operator
- An incremental implementation that uses a recursive algorithm without manual incrementalization

```

1 public void Initialize() { }
2 public double ComputeAverage() {
3     return root.Items.AsEnumerable().
4         Average(i => i.Value);
5 }

```

Listing 5 Computing the average value from a list of items using query operators

The batch implementation that recalculates the average after every model manipulation is depicted in Listing 5. For this implementation, we inserted a call to `AsEnumerable` which simply forces the compiler to use the regular query API in order to avoid the runtime compilation of predicates.

```

1 public void Initialize() {
2     resultCache = Observable.
3         Expression(() =>
4             root.Items.Average(i => i.Value)
5         );
6 }
7 public double ComputeAverage() {
8     return resultCache.Value;
9 }

```

Listing 6 Incrementally computing the average value from a list of items using query operators

The incremental version of this solution is depicted in Listing 6. It uses the NMF model query API that is manually incrementalized. It starts the incremental calculation of the average computation once in the `Initialize` method using the `Observable.Expression` method. This method turns the lambda expression that it receives as an argument into a DDG. During the construction of the DDG, the incrementalization of the call to the `Average` method uses the provided manual incrementalization which in turn makes use of the incrementalization system to convert its inner function argument `i => i.Value` into a DDG template. Afterward, this version only gets the current result through the `Value` property of the incremental result object. Any updates of the DDG are performed through hooks of the model manipulation.

The implementation without a manually incrementalized function is depicted in Listing 7. Because iterators have a state, we cannot use the .NET collection interfaces and therefore use recursion to loop through the collection indices. Unlike the query API, this assumes that the collection in question is orderable and therefore has an index-based access (which collections in NMF often do not have). The recursive function starting in line 6 takes an index and a tuple of a running sum and element count. If the index is smaller than the number of elements in the collection (cf. line 8), the function recursively calls itself with the successor index and adds the value of the current item (lines 9–10), otherwise the current tuple of running sum and element count is returned (line 11). Another function (lines 2–4) then takes this tuple and

computes the final result by dividing the running sum by the element count.

```

1 public void Initialize() {
2     var computeAverage =
3         Observable.Func(((int, int)
4             tuple) =>
5             ((double)tuple.Item1) / tuple.
6                 Item2
7         );
8     var recurse =
9         Observable.Recurse((rec, index,
10             tuple) =>
11             index < root.Items.Count
12             ? Add(rec(index + 1, tuple),
13                 root.Items[index].
14                     Value)
15             : tuple
16         );
17     resultCache = Observable.
18         Expression(() =>
19             computeAverage.Evaluate(
20                 recurse.Evaluate(0, (0, 0))
21             )
22         );
23 }
24 private static (int, int) Add((int,
25     int) before, int value) {
26     return (before.Item1 + value,
27         before.Item2 + 1);
28 }
29 public double ComputeAverage() {
30     return resultCache.Value;
31 }

```

Listing 7 Incrementally computing the average value from a list of items using a recursive algorithm

Note that lambda expressions in C# by default do not support recursion as the lambda expression cannot be referenced before it has been declared. Our implementation solves this problem by a dedicated `Recurse` method that incrementalizes the iterative fixed point of a given function. In particular, the first argument of the function passed into `Recurse` in line 6 is itself a function that allows callers to make a recursive call.

7.2.2 Results

The performance measurements for the recalculation of the average are depicted in Fig. 6. All measurements were taken on an Intel i7-8550U CPU clocked at 1,80Ghz in a system with 8GB RAM. The sizes refer to the number of `Item` elements in the generated model.

For the recursive algorithm, we were only able to run the benchmark up to a size of 1000 items as the benchmark ran into a stack overflow for larger model sizes. This recursion

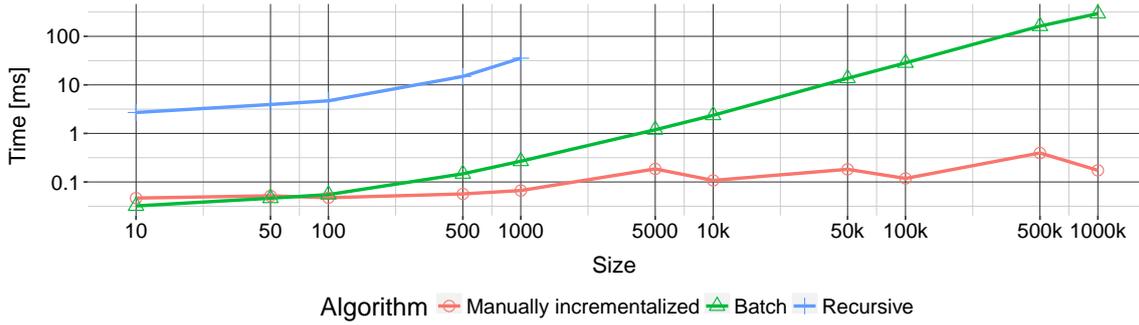


Fig. 6 Performance results of the micro-benchmark to compute averages

happens when attaching the DDG to the model as attaching nodes iteratively implies attaching of dependent nodes.

One can see that starting at a given size, the slope for the batch results is constant. This is because, from this point, the time to calculate the average is completely determined by the number of items to process.

The recursive algorithm eventually has the same slope but it takes slightly longer. This is because the recursive algorithm has to process all items starting from the collection index where a change was made. This means, if an element has been added at the end of the collection, then the change propagation can be done in constant time since the calculation of sum and element count is still valid up to the previous index. If the value of an existing item is changed, then the change propagation has to recalculate the recursion from that index onward, resulting in a linear effort. However, because the DDG adds a lot of overhead and defeats the compiler optimizations, calculating the average incrementally using the recursive algorithm is about two orders of magnitude slower than just recalculating it every time it is required.

For the manually incrementalized version, the effort to propagate any change is simply constant. For a new item, a new DDG node is created to track its value which causes to raise a notification that a new value has been added for the average calculation. When the value of an existing item has changed, this causes a notification that a value has been replaced. In both cases, the running sum and element count can be updated in constant time. However, the slope is not as clear as for the other curves and it is not completely flat. For this, we blame caching issue because for all larger model sizes, not all elements fit into the CPU caches. In addition, the accuracy for time measurements below a millisecond is not ideal, even though we are using hardware performance counters. Nevertheless, the slope indicates an almost constant change propagation time which is much faster than the batch recalculation for large models. For models with a million elements, we already see that the incremental version is more than three orders of magnitude faster than the batch algorithm.

7.3 The TTC 2015 Train Benchmark

In this section, we analyze the Train Benchmark case [19] at the TTC 2015 to which an NMF solution was submitted [42]. This benchmark consists of five analysis queries. The only incremental tools that participated in this contest were EMF-INCQUERY from the case authors and the NMF solution.

7.3.1 Benchmark setup

In the scope of this case study, we only investigate the TTC version of the Train Benchmark [19] that covers a subset of the entire Train Benchmark [43]. We briefly describe the benchmark setup here, but further details can be found in these papers.

Besides the *SwitchSet* query briefly introduced in Sect. 2, the benchmark included four other queries: *PosLength* queries the segments in the railway network that have length 0 or less. *SwitchSensor* looks for switches without a corresponding sensor. *RouteSensor* looks for switch positions along a route that refer to switches that are not part of a sensor that is defined within that route. The most complex *SemaphoreNeighbor* query searches for routes that end at a given track segment (which means that the next track segment belongs to a different route), but the exit semaphore of the route is not the entry semaphore of the route that starts with the connected segment.

The benchmark consists of four phases depicted in Fig. 7: *Read*, *Check*, *Repair* and *Recheck*. In the *Read* phase, solutions of the benchmark load the model of the respective size. In the *Check* phase, the number of invalid elements is computed, i.e., the number of pattern matches where each pattern match represents a constraint violation. In the *Repair* phase, several of these (either 10 or 10% of all, determined by the parameter *Change set size*) constraint violations are fixed. After that, the number of invalid elements is refreshed in a *Recheck* phase of the benchmark. For each combination of input model (size) and change set size, the benchmark is run five times for each solution. Within each run, phases *Repair* and *Recheck* are repeated ten times.

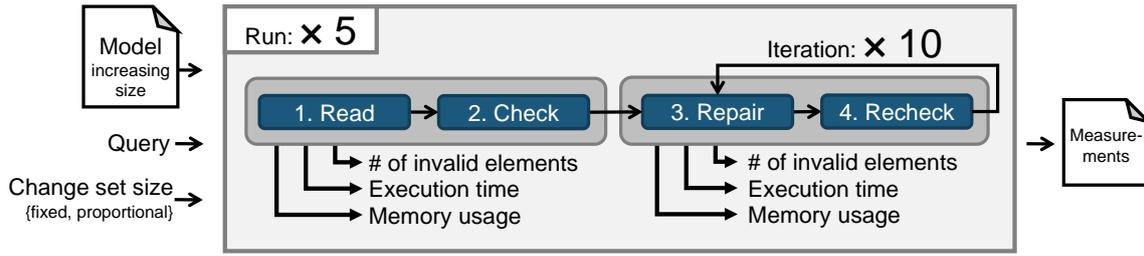


Fig. 7 Phases of the Train Benchmark [19]

```

1 Fix(rc.Descendants().OfType<Segment
2   >()
3   .Where(seg => seg.Length <= 0)
4   ,
5   action: segment =>
6     segment.Length = -segment.
7     Length + 1);

```

Listing 8 NMF implementation of the *PosLength* query

During the execution of these phases, the benchmark collects metrics on execution time, number of invalid elements and memory usage.

In the scope of this case study, we are specifically interested in incremental revalidation, i.e., the time for *Repair* and *Recheck*.

7.3.2 The NMF solution

The description of the NMF solution to the Train Benchmark is based on the original submission to the TTC 2015 [42].

In the solution, we use NMF Expressions to incrementally query the source model and cache the invalid elements continuously. However, this means that the phases *Repair* and *Recheck* get merged as the model manipulation automatically updates the incremental query result. In particular, the *Recheck* phase becomes meaningless as the updated results are always available and could be used for immediate feedback, while more computational effort is put into the model manipulation tasks in the *Repair* phase.

In the following, we present the solution to the tasks, following the structure of the case description, though with omitted sort keys. The explicit parameter name *action* is not necessary but is inserted to improve readability.

The implementation of the *PosLength* query is depicted in Listing 8. It uses the *Descendants* operation of NMF to iterate overall models contained somewhere in the railway container. To this collection of model elements, a type filter is applied that restricts the collection to instances of *Segment*. This collection of segments is finally filtered to those that have a length below 0. The repair operation sets the length as suggested in the case description 2.

```

1 Fix(rc.Descendants().OfType<Switch
2   >()
3   .Where(sw => sw.Sensor == null
4   ),
5   action: sw =>
6     sw.Sensor = new Sensor());

```

Listing 9 NMF implementation of the *SwitchSensor* query

```

1 var routes = rc.Routes
2   .Concat(rc.Invalids.
3   OfType<Route>());
4 Fix(from route in routes
5   where route.Entry != null
6     && route.Entry.Signal ==
7     Signal.GO
8   from swP in route.Follows
9   where swP.Switch.CurrentPosition
10  != swP.Position
11  select swP,
12  action: swP =>
13    swP.Switch.CurrentPosition =
14    swP.Position);

```

Listing 10 NMF implementation of the *SwitchSet* query

The implementation of *SwitchSensor* is depicted in Listing 9 and works very similar, though this time, elements of type *Switch* are selected and filtered for those where no sensor is attached. The repair operation creates a new sensor and assigns this to the *Sensor* property of the selected switch. Note that because *Sensor* is a container property, this moves the switch out of the model.

The implementation of the *SwitchSet* query was already explained in Sect. 2. We depict it again in Listing 10. This query (and all of the below) is based on a collection of routes. Because routes can only appear in two places, namely their correct location in the containment hierarchy and in the *invalids* reference, we make this more explicit to only look in these two places than traversing the entire containment hierarchy. Note that this does hardly make any difference for the incremental runtime because the containment hierarchy of the model is not touched in most queries.

```

1 Fix(from route in routes
2     from swP in route.Follows
3     where swP.Switch.Sensor != null
4         &&
5         !route.DefinedBy.Contains(swP.
6             Switch.Sensor)
7     select new { Route = route,
8                 Sensor = swP.Switch.Sensor
9     },
10    action: match =>
11        match.Route.DefinedBy
12            .Add(match.Sensor))
13 ;

```

Listing 11 NMF implementation of the *RouteSensor* query

The repair operation of the *SwitchSet* query sets the current position of the switch to the position required by the route.

The implementation of *RouteSensor* is depicted in Listing 11. It iterates through all routes and all switch positions defined by these routes and selects those where sensor of the corresponding switch is not defined in that route. The repair action adds the sensor to the collection of sensors of that route. Because *DefinedBy* is a containment reference, this again moves the sensor within the model.

The implementation of the *SemaphoreNeighbor* query is twofold as depicted in Listing 12. We use a helper function to compute the route that follows the current route of a route provided as input. For this, we iterate through the sensors of the route, iterate through all of its track elements, iterate through the connected elements and to those where the connected element is defined in a different route than the current one. The railway network allows at most one of such next routes. In the actual pattern, we then iterate through the routes, find the candidate for the next route and save it as a local variable, then filter this pair of routes to check that the entry of that route is not the same as the exit of the first route. The repair operation sets the entry of the second route to the exit semaphore of the first route.

The solutions to *SwitchSet*, *RouteSensor* and *SemaphoreNeighbor* use the query syntax of C#. This syntax is translated to the method chaining syntax by mapping the query keywords like *from* or *where* to SQO method calls supported by NMF Expressions. The *let* expression in the *SemaphoreNeighbor* query is converted to a *Select* method that maps a route to a pair of routes, represented by an anonymous type.

Because NMF Expressions allows to use the same specification both in a classic batch manner as also incrementally, our solution can also be configured to run in batch mode without any changes to the patterns. When executed in batch mode, NMF Expressions forward the call to the LINQ to objects implementation. Besides a negligible runtime com-

```

1 var connectedRoute =
2     ObservingFunc<IRoute, IRoute>.
3     FromExpression(
4     route =>
5     (from sensor1 in route.
6         DefinedBy
7         from tel in sensor1.Elements
8         from te2 in tel.ConnectsTo
9         where te2.Sensor != null &&
10            te2.Sensor.Parent !=
11            route
12         select te2.Sensor.Parent as
13             IRoute)
14     .FirstOrDefault());
15
16 Fix(from routel in routes
17     let route2 = connectedRoute.
18         Evaluate(routel)
19     where route2 != null && route2
20         .Entry != routel.Exit
21     select new { Route1 = routel,
22                 Route2 = route2 },
23    action: match =>
24        match.Route2.Entry = match.
25            Routel.Exit);

```

Listing 12 NMF implementation of the *SemaphoreNeighbor* query

pilation effort, this utilizes the highly optimized platform LINQ implementation.

The patterns are *enumerable expressions* where developers can choose at runtime whether the pattern should be executed in batch mode or whether NMF Expressions should register for elementary change notifications to keep a cache of the result up to date. To specify patterns, we created a small method *Fix* that captures them.

The easiest implementation for the *Fix* function repairing any validation error as soon as they occur would be the one presented in Listing 13. In Line 2, we tell NMF Expressions that we want to obtain incremental updates for the given pattern. Line 3 repairs all occurrences existing so far and Lines 4-8 handle new pattern matches. For the benchmark, we adopted the *Fix* function to account for the benchmark phases. In particular, the implemented version takes a third parameter to allow us to sort matches. Since these sort keys offer little insight, we omit them in the pattern presentation.

7.3.3 Results

The results from the open peer reviews²¹ are depicted in Table 1.

²¹ <https://docs.google.com/spreadsheets/d/1WepbTGB8XbXFV6tYK DsdOn9kFvai8c4q\EoszeV3FsI/edit?usp=sharing>.

```

1 public void Fix<T>(
    IEnumerableExpression<T> pattern
    , Action<T> action) {
2     var patternInc = pattern.
        AsNotifiable();
3     foreach (T element in patternInc)
        {
4         action(element);
5     }
6     patternInc.CollectionChanged += (o
        ,e) => {
7         if (e.NewItems != null) {
8             foreach (T element in e.
                NewItems) {
9                 action(element);
10            }
11        }
12    }
13 }

```

Listing 13 A simplified implementation of the Fix function

Table 1 Results from the open peer review of the TTC 2015 Train Benchmark

Tool	Correctness and completeness	Conciseness	Readability
ATL/EMFTVM	15	12	13
EMF-IncQuery	12.5	12.5	12.5
FunnyQT	15	15	12.5
NMF	12.7	13.3	15
SIGMA	15	13.3	13.3

For the understandability, the NMF solution was the only solution at the TTC 2015 contest that received full points for understandability from all open peer reviewers. In particular, the solution was evaluated to be more understandable than all batch solutions written in FUNNYQT [44], ATL/EMFTVM [45] or SIGMA [46].

The performance measurements for the revalidation for all queries are depicted in Fig. 8. All measurements were taken on an Intel i7-4710 CPU clocked at 2.50 Ghz in a system with 16GB RAM. We use the same size notation as in the Train Benchmark where a size 1 corresponds to about 1300 model elements. In the largest considered size 1024, the model contains about 1.5 million model elements.

The NMF solution supported two execution modes, incremental and batch mode. In the batch mode, the analysis is rerun on the entire model in each step whereas the incremental version maintains intermediate results and invalidates them whenever elementary changes appear in the model.

The results show that the incremental version of the NMF solution was able to keep up with specialized tools such as EMF- INCQUERY for many model sizes and queries and even

beat EMF- INCQUERY by roughly a magnitude in the *SwitchSet* query that we used in the motivational example. In other scenarios, our implementation eventually gets slower than the EMF- INCQUERY solution.

Figure 8 also shows that for model sizes up to roughly 100,000 model elements (size 64), the incremental NMF solution was the fastest for all queries depicted.

To explain why the NMF solution is faster than EMF- INCQUERY for the *SwitchSet* query, we depicted the Rete network created by EMF- INCQUERY for the *SwitchSet* query introduced as running example for this paper in Fig. 9. In the running incremental analysis, each node depicted in this graph represents a list of partial pattern matches. The network combines simple references and attribute accesses through *Join* nodes to pattern matchers. On the contrary, nodes in the DDG created by NMF Expressions only represent an evaluation of an attribute or reference for a single model element.

Besides the different granularity, the path in the data structure for a given change is different. In the NMF solution, a change in a signal position only affects four nodes—the respective attribute evaluation node, the binary expression node that checks equality, the node for the *where* operator and lastly the node for a compiler-generated *select* operator. Meanwhile, the same change in the Rete network depicted in Fig. 9 has to be propagated through seven nodes. In NMF Expressions, every node deals with exactly one value and therefore has deterministic constant cost. In EMF- IncQuery, nodes process in general more data. Through the usage of hash tables, the nodes still have a constant effort, but only amortized.

The results for the batch validation are depicted in Fig. 10. Both NMF solutions had a relatively low constant overhead, indicated by the fact that the solutions were much faster than EMF- INCQUERY or Java solutions. For larger models, the overhead of the incremental algorithm to set up caches for immediate results and register event handlers is similar to the query effort done in the batch mode.

In the *SemaphoreNeighbor* query, the overhead of creating the DDGs for the incremental revalidation is slightly higher than in the other cases due to the higher complexity of the query.

The incremental version has a drawback against the batch version: memory consumption. We experienced that the incremental version did not allocate more memory than the batch version (since batch analysis has to reallocate memory for each evaluation run) but the DDG that is responsible for most of the memory consumption is continuously required and cannot be released until it is detached from the model. When the analysis is run in batch mode, the memory allocated to compute the analysis can be released once the analysis result is available.

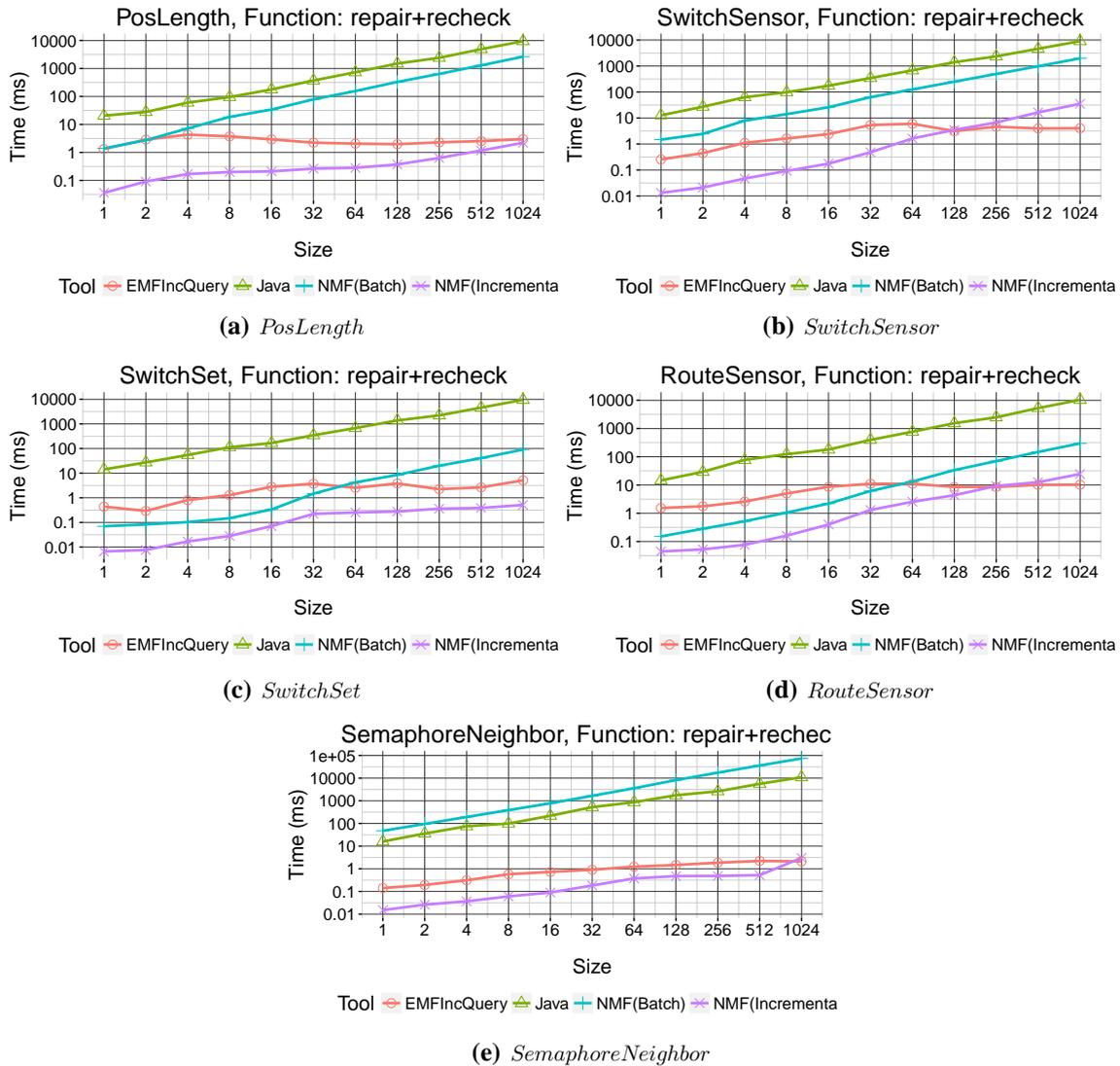


Fig. 8 Performance results for revalidation. The graph compares the NMF solution (batch and incremental mode) with the reference solutions in Java and EMF-IncQuery. Both axes are logarithmical

However, since both the .NET runtime and the Java Virtual Machine employ garbage collection, the memory consumption is difficult to measure because memory no longer used may still be allocated because the exact time of a garbage collection is not known. To at least get an impression on the memory consumption, we depicted the working set of the benchmark queries in Fig. 11.

The results show that the working set was within half an order of magnitude difference as the Java or EMF-INCQUERY solution, at least for the queries *SwitchSensor*, *SwitchSet* and *RouteSensor*. The working set also remained within the limit of 2GB even for the largest models which is why we think that the memory requirements are feasible for this scenario.

For the *PosLength* query, the memory consumption is very high, because of the large amount of segments that are con-

tained in the model. For each segment, a DDG to check whether its length is less or equal to zero has to be instantiated. On the contrary, the model manipulation performed for the *PosLength* query is computationally inexpensive. Therefore, the overhead due to incremental computation is relatively higher.

For the *SemaphoreNeighbor* query, this result is different. Rather, the memory consumption of the incremental execution mode is about an order of magnitude higher than the memory consumption of the batch mode execution and all solutions required much larger amounts of memory. We guess that this is due to the different query operators used, as especially the *SelectMany* operator turns out to be very memory intensive in the incremental setting.

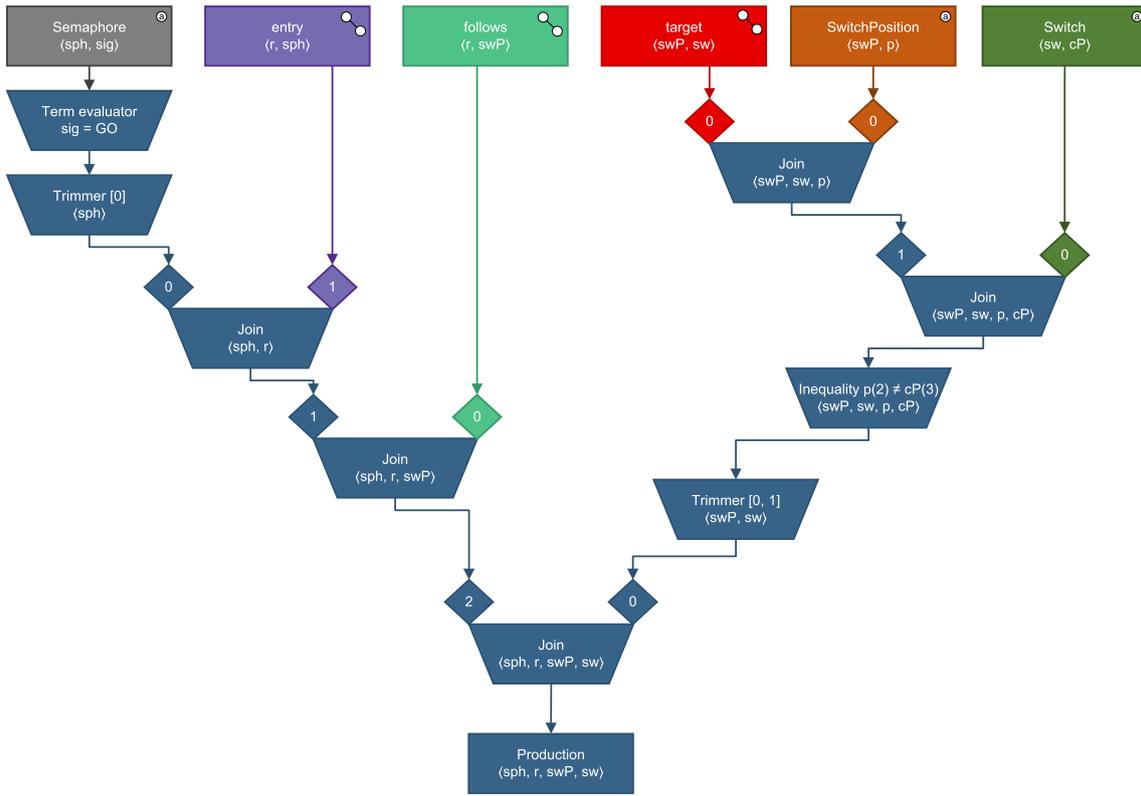


Fig. 9 Rete network created by EMF- INCQUERY for the *SwitchSet* query of the Train Benchmark (cf. [47])

7.4 Threats to validity

In this section, the validity of the results obtained in the presented validation is discussed. We separate this discussion in the internal validity in Sect. 7.4.1 and external validity in Sect. 7.4.2.

7.4.1 Internal validity

The internal validity of the case study results depends on the type of result. In particular, there is a huge difference between runtime measurements, memory measurements and questionnaire results.

We can safely exclude an experimenters bias for the Train Benchmark as the case has been authored by other researchers and—more importantly—the set evaluation criteria have undergone a peer-reviewed process. This does not hold for the micro-benchmark for computing averages.

Performance There is a threat of confounding factors for the performance measurements. Other applications than the benchmark may be running on the machine such that the measured performance times may not be completely accurate.

To compensate this threat, all background services have been terminated where possible, including messengers, storage services and connection services.

However, there are also some services that are inevitably connected to the benchmarks such as the garbage collector and the just-in-time compiler. To reduce the influence of the latter two, all measurements have been repeated ten times for the Train Benchmark and fifty times in the micro-benchmark.

The benchmark framework of the TTC Train Benchmark 2015 does not consider an elimination of the just-in-time compiler such that this compilation does influence the results. However, this influence is only important for the smaller model sizes. For the larger model sizes, the time for the just-in-time compilation can be neglected. In the micro-benchmark, we tried to eliminate this influence by warming up the system, i.e., we ran the benchmark without measuring results.

For the TTC Train Benchmark, less on the measurement itself, there is also a difference of the used technology because other solutions generally use EMF instead of NMF. Therefore, differences in response times may be due to the difference of the used framework instead of difference in the used incremental tool. Clearing this effect from the measure-

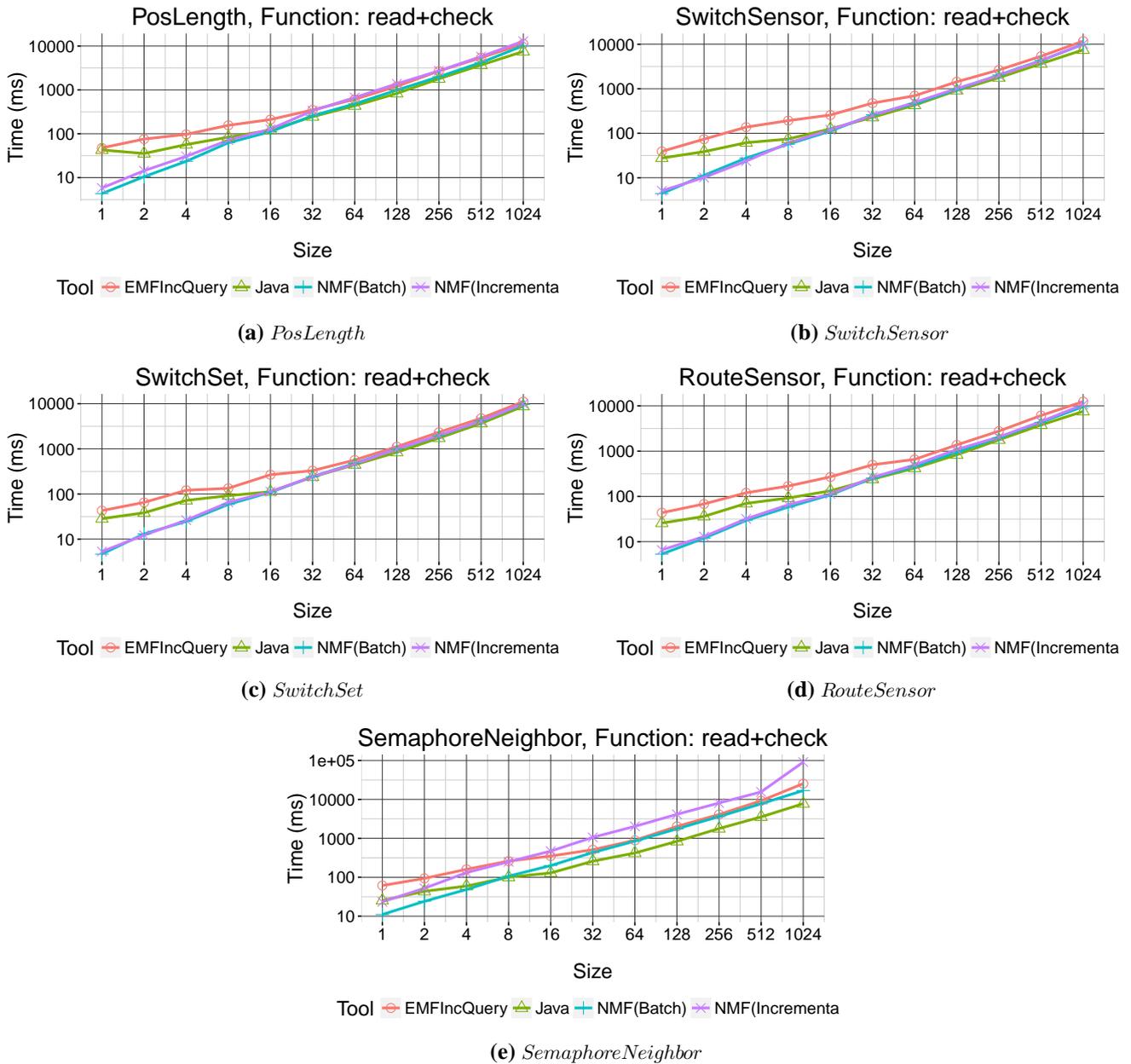


Fig. 10 Performance Results for batch validation of the NMF solution versions compared to the reference solutions in Java and EMF- INCQUERY (both axes logarithmic)

ments required to reimplement the tools in another platform, an overhead which is not justified by this confounding effect.

Due to repetition of measurements, we think that the influence of garbage collection and just-in-time compilation is much smaller than the observed differences between incremental and non-incremental tools.

Memory Measurements For the memory measurements, there is a large confounding factor because we only measured the working set size. Therefore, the memory measurement is confounded by the memory consumption of the modeling

framework as well as the memory consumption of any infrastructure code. Lastly, the memory consumption also depends on the memory efficiency of the underlying technology which is often different because NMF uses .NET meanwhile other solutions usually use Java.

Furthermore, the garbage collector is a very important confounding factor for the memory measurements because it makes a tremendous difference whether the memory measurement is done before or after the garbage collector frees memory for unused objects. Because it is not possible in .NET to clearly identify when garbage collection has taken

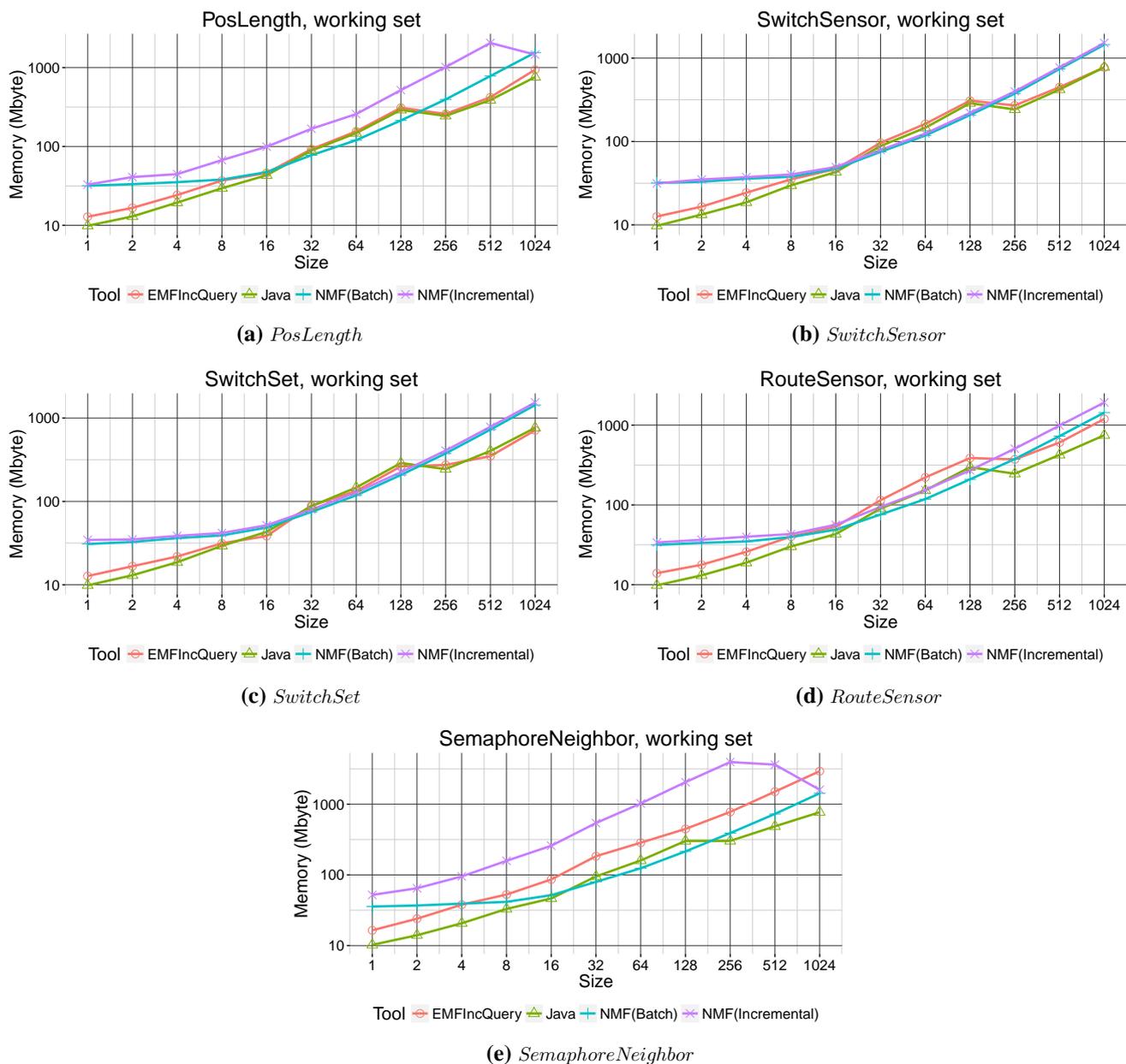


Fig. 11 Working sets in the Train Benchmark against relative model size (both axes logarithmic)

place or to manually trigger it,²² the influence of garbage collection cannot be avoided.

Therefore, what one would rather want to measure is the amount of memory that is actually used by the incremental tool. However, this is not possible easily with the current architecture. Therefore, the memory measurements have a low accuracy.

Understandability This threat only concerns the Train Benchmark since understandability has not been evaluated for

²² It is indeed possible to *suggest* the system to perform a garbage collection but it is not guaranteed when that happens.

the micro-benchmark. We list and discuss the most important threats to internal validity of the understandability below.

Confounding factors There are severe confounding factors in the data for the understandability: NMF is yet a relatively unknown approach and in general, the C# language is much less common than, for example, Java in the model-driven community. Therefore, many participants of the TTC are (sometimes even admittedly) not familiar with the technology, which clearly confounds the understandability results.

History For the open peer reviews, it is unclear in which order the assigned solutions were reviewed. Therefore, an influence of history cannot be excluded. For the presentation at the TTC, there is a clear influence of history since the solutions are presented in sequence.

Instrument change An instrumentation effect can be excluded. Open peer reviews have been asked for by the TTC organizers of 2015, not by the authors of this paper.

Several common forms of internal threats to validity such as differential attrition, ambiguous temporal precedence, maturation, diffusion or regression toward the mean do not apply because the understandability was only evaluated once and not over a period of time.

Overall, the results for understandability are very inaccurate and have to be taken with great care. Mainly for the unfamiliarity of the TTC audience both with NMF and .NET in general, we think that the true understandability of the NMF solutions is better than the understandability from the questionnaire responses.

7.4.2 External validity

Again, we split the discussion of external threats to validity on the type of validations. However, since both are hard metrics, the threats to external validity are the same for performance and memory measurements.

Performance and Memory Measurements For performance and memory measurements, we face the problem that it is unclear to what degree the obtained results can be generalized for other applications, input model characteristics and change sequences. Further, it is unclear to what extent the opponent solutions represent the opponent tools, even though many opponent solutions have been authored by tool authors.

Though the change sequence used in the various case studies have been generated, they depend on the selection of changes and their proportion. In the evaluation performed for this paper, we aligned to the original benchmark proposal that only consists of homogeneous change sequences. This is required to allow a comparison to other tools, but limits the external validity of the results.

For the comparison with EMF-IncQuery, we note that the EMF-IncQuery solution to the TTC Train Benchmark 2015 was written by the authors of EMF-IncQuery. Therefore, we can safely assume that this solution is the best solution possible using this tool.

Understandability The participants of the TTC cannot be seen as representative for the likely users of incremental model analyses and incremental model transformations. Rather, in the last years, they represent the authors of solutions to the TTC cases and perhaps a few other participants of the STAF event in which the TTC is embedded. The solution authors may or may not be biased toward their own solution,

even though all of them are researchers and therefore should give an unbiased opinion on all solutions.

A similar statement holds for the open peer reviews where reviewers may or may not be biased toward their own solutions. Because in the open peer reviews of the TTC, each solution is only reviewed by two reviewers, there is an influence of chance whether the reviewers are biased.

However, the selection of reviewers and the selection of participants of the solution presentations at the TTC is outside the control of the author of this thesis. Therefore, similar to the validation of correctness, the threats to validity limit the expressiveness of the validation data but as we cannot influence the validation setup, we do not have an option to make the data more conclusive.

Overall, we think that the results on the understandability are rather preliminary and should be supported by future research to gain credibility.

7.5 Summary

The micro-benchmark about computing averages of a collection has proven that manual function incrementalizations can substantially improve the performance of incremental analyses, both in terms of memory, response times to changes and also the maximum model sizes that can be processed. Whereas the implementation of a recursive algorithm is substantially slower than just recalculating the average every time, the manually incrementalized version has a better asymptotic complexity. For the largest model size considered, we have that it is more than three orders of magnitude faster than the batch algorithm.

The queries and repair transformations demonstrate a good conciseness of the C# language for the Train Benchmark. We think that it is difficult to get a more concise textual solution for this case and this opinion has been confirmed by a very good evaluation of the understandability for the NMF solution. At the same time, developers get the full tool support from, e.g., Visual Studio and the query syntax that we use is used by thousands of developers already and widely understood.

The performance figures show that the incremental version of our solution outperforms the batch mode execution of the same solution in all cases, often by multiple orders of magnitude. Compared to the incremental pattern matching tool EMF-INCQUERY, we see that the performance is about as good for most of the queries. Especially for medium-sized models, the revalidation times are better for all queries.

Another advantage of our solution is that it gives both a batch mode solution and an incremental solution of the same pattern specifications. Thus, the same analysis code can be used in the case setting where incrementality is a clear advantage, or in a batch mode, e.g., when memory is a sparse resource or the analysis results are only required once.

8 Related work

This section reviews the current state of the art in incremental computation. We divided the existing approaches into general-purpose approaches applicable to any analysis through support of a Turing-complete language (and their relatives in reactive programming) and those specific to a class of analysis. We refer to the latter as specialized incremental tools. These approaches do not have a restriction in the domain but in the kinds of analysis that are supported, e.g., only queries or graph patterns.

8.1 General-purpose incremental computation systems

Pugh and Teitelbaum [48] were the first to apply memoization to incremental computation. Memoization is applicable to any referential transparent function but rests on the assumption that the data structures it operates on is immutable—an assumption often not met for modeling frameworks, in particular, not for EMF or NMF. Immutable data structures cannot represent cyclic data structures as easily and therefore make it very difficult to create analyses requiring them. In the running example, the analysis would require the railway container in order to connect switch positions to switches. The latter, however, changes with every model change, making memoization just a waste of memory. In general, it is unclear which functions should be memoized to actually get a performance benefit.

Later, Acar and others created Self-Adjusting Computation (SAC), a framework to support the development of incremental programs [36] using the then newly introduced DDGs. A good overview on SAC is provided by Acar [49]. The rough idea is to memoize the computation made for a given analysis. While the framework originally supported functional languages, it has been extended to imperative languages based on C [50]. SAC operates on the batch specification of an analysis annotated with explicit incrementalization primitives. From these primitives, a DDG is deduced that keeps track of changes.

Closest to our approach, Traceable Data Types (TDTs) have been integrated into SAC to allow developers to supply a custom incrementalization of an algorithm in a dedicated data structure [33]. TDTs have an internal virtual clock and work by allowing developers to explicitly revoke previous operations and return the earliest inconsistent state, if any. These operations include both state management and queries, which allow different states during an analysis, but require the developer of an analysis to manage the state of the data structures on their own. As a consequence, TDTs are limited to their own data structures while our approach allows the incrementalization of higher-order methods that are independent of data structures used in predicates. Furthermore,

TDTs require some boilerplate code to use them in SAC [33]. Our implementation only requires a method annotation so that the dynamic algorithm can be reused in its compiled form.

Based on SAC, Carlsson was the first to find that incrementalization can be represented as a monad [23]. Carlsson used monads (as a Haskell language feature) to realize explicit incremental computing on immutable data structures. This is in contrast to our work, as we explicitly consider state and are therefore able to operate on mutable models, which in our view better reflects the nature of models. Because Carlsson requires the user to insert boundaries of the monad explicitly, the approach is able to operate on functions as black boxes. In contrast, we operate implicitly. On the downside, this means that we require models of the analysis (its AST) and the used operators. However, these models are worthwhile as they allow us the integration of dynamic algorithms while the correctness is still covered by Theorem 1.

Further, because we do not use monads as a language feature for the implementation, we refer to it more as a term from category theory. In particular, as argued in this paper, incrementalization is conceptually *not* a monad (in the original meaning from category theory) because the unit transformation must not be natural—a fact that Carlsson ignores because it is not strictly required for an implementation in Haskell.²³

Hammer and others introduced the idea of demanded computation graphs, implemented in Adapton [15,16]. Demanded computation graphs make sure that a change propagation is only performed if the result is actually needed, a feature that we implemented using reference counters on the DDG nodes. Similar to SAC, Adapton does not work implicitly, such that developers have to think carefully about where to insert incrementalization primitives.

SAC and Adapton both have the problem that they work explicitly: The programmer must give a hint which parts of an analysis can be saved for repeated analysis runs. Furthermore, the mutation of the input must be done through a dedicated mutator component in SAC or through refreshing computation thunks in Adapton. Our approach is able to pick up change notifications from the generated model API and works entirely implicitly, i.e., the programmer does not have to change the code at all. This makes it possible to use incrementalization in mainstream technologies.

Chen et al. developed an approach to transform programs written in purely functional Standard ML into SAC [9], allowing developers to omit incrementalization primitives. Hence, the approach works implicitly. However, we are not aware of any research that integrates the usage of TDTs into this framework to make them more efficient. Therefore, this

²³ In the context of functional programming, monads are often defined without the requirement that the unit transformation must be natural.

technique has the problem that incrementalizations of collection operators are inefficient.

On a rather technical level, neither Adapton nor SAC are currently available to be used with MOF models. While there is no publicly available implementation of SAC, Adapton has a freely accessible and maintained implementation in Rust.²⁴ However, Rust has very limited support for object-oriented design. In particular, Rust only allows inheritance and dynamic binding for traits, but trait objects cannot be downcasted. However, this is a mandatory requirement for many benchmarks such as, e.g., the Train Benchmark.

Other approaches to incremental computation include entirely new programming models that allow an easy incrementalization or parallelization. An example of these approaches is revision-based computing [51]. However, here the developer has to explicitly think about where to insert such a revision concept.

IceDust [52] and its successor IceDust 2 [53] are systems that allow users to use incremental computations through a system of derived features. For each derived feature of a class, the user can decide whether changes affecting it should be computed incrementally (which means immediately after the source features have been changed), on-demand (when the value is read) or eventually (on a different thread). In contrast to our approach, we do not see IceDust making use of dynamic algorithms to speed up the recomputation of analysis results. On the contrary, we have no experience using our approach for derived features.

8.2 Reactive programming

A very related paradigm to incremental computation is reactive programming where the goal is to get notifications for changes. An overview of 15 languages for reactive programming was created by Bainomugisha [54]. According to the taxonomy suggested in this survey, our implementation NMF Expressions is based on events with a push-based evaluation model and implicit lifting.

Our approach can be interpreted as a way with a formal foundation how implicit lifting can be overridden to gain performance. Most approaches in reactive programming circumvent this problem as they apply explicit lifting [54]. Even if the lifting works implicitly, the approaches do not incrementalize methods using their structure but rather execute a given predicate as is.

In reactive programming, the developer has to explicitly declare signals to which the analysis should react. Our approach makes this implicit as the incremental algorithm automatically attaches to the model as required. Notable exceptions are FRTIME [55] and FLAPJAX [56].

²⁴ <http://adapton.org/>, retrieved 18 Jul 2017.

Particularly on the .NET platform, Reactive Extensions (Rx) have been introduced to support reactive programming [57]. Similar to our approach Rx uses the query syntax to combine several streams of data.

Reactive programming approaches are built upon an important assumption, namely that signals do not change once they are processed. That is, they operate on an (potentially infinite) sequence of immutable data. This is a contrast to model analysis where the model usually has an approximately constant size, but is mutable.

8.3 Specialized incremental approaches

Incrementality is a desirable property as it promises to save computational effort when analyses are computed repeatedly. Therefore, it has been a subject of research for decades [58], prominently, for example, with the search for incremental compilers [59]. Common to all of these approaches is that they make advantage of abstractions they make on the analysis to perform at the cost of limited applicability. As soon as one formalism alone is no longer capable of expressing an analysis, multiple approaches must be integrated, causing a large integration overhead [60]. This is especially important in maintenance scenarios where perfective changes require to extend an analysis such that it no longer suits the given formalism.

Among the first incrementalization systems specialized on a limited class of problems is the approach by Reps [61] for attribute grammars. This approach works by using a static dependency graph for attribute evaluations for which Reps has shown that an optimal-time reevaluation strategy can be found by reevaluating the attributes in a topologically sorted order of a static dependency graph. This approach rests on the assumption that the data processed by the attribute grammar is immutable. As Reps applies this technique for parse trees, this assumption is reasonable, but it does not hold for models in general.

Willis et al. achieved convincing results for an implicitly incremental execution of the Java Query Language [10]. In their approach, they find all the places where caches may be invalidated through aspect-oriented programming. As a consequence, all these places must be known at compile-time. Thus, model manipulation and analysis are tightly coupled and cannot be separated into different modules.

On the .NET platform, a range of non-academic projects aimed to provide change notifications for queries, sometimes with an incremental execution. Among the rather mature are Continuous LINQ [62], Bindable LINQ²⁵ and Obtics.²⁶ However, the example query we present in Sect. 2 only works with Bindable LINQ where

²⁵ <http://bindablelinq.codeplex.com/>.

²⁶ <http://obtics.codeplex.com/>.

the runtimes are far worse than rerunning the query for each elementary change.

A similar problem to the incremental queries appears in relational databases when maintaining materialized views [63]. An overview on the research can be found in [64]. These approaches have also been applied to object-oriented databases, as, for example, in [65]. Some approaches like notably LINQ or SQUOPT [66] have ported this database technology to object-oriented languages, in case of SQUOPT Scala.

A popular approach to specify queries, especially in graph transformation, are Graph Patterns. Bergmann et al. created INCQUERY, an incremental pattern matching system for Graph Patterns [11,12]. This approach uses a Rete network [67], a static dependency graph, whose nodes are primitive filter conditions or joins of partial pattern matches. Each node represents a set of (partial) pattern matches. This approach can support mutable models because the notification API of models can be used to determine when matches must be revoked or new matches arise. Unlike NMF Expressions that requires a *dynamic* dependency graph, this means that queries as in the Train Benchmark can be incrementalized using only a *static* dependency graph.

INCQUERY was integrated to EMF models as EMF-INCQUERY [68]. This system was also used to evaluate queries [69] and certain OCL expressions [70].

An incremental execution of OCL has also been subject of research for incremental constraint checking [71–73]. These approaches are either limited to boolean-valued constraints or limited to static analysis.

Lastly, in the field of algorithmics, a whole class of algorithms is dedicated to process incremental changes, dynamic algorithms. Prominent examples from this field include the dynamic spanning forests by Holm et al. for connectivity analysis [31] and King and Sagerts approach for dynamic transitive closures [74]. These algorithms often have a sub-optimal runtime in absence of changes but can update their data structures according to changes. Our approach does not compete with these dynamic algorithms but provides a way how they can be used to specify incremental analyses using a batch specification.

A generic theory of changes applicable in a wider range of applications was developed in the scope of the SCUOPT project by Cai et al. [37]. While this approach has a wider applicability than just a single class of analyses, the authors do not yet have a concept how to mix several of such analysis kinds. Thus, it serves as a foundation for the development of specialized incremental tools. In the paper, Cai et al. applied the approach to a map-reduce technique, which is a small subset of incremental queries.

9 Conclusion

In this paper, we have introduced a novel approach to formalize incremental computation systems as functors from category theory (C1). This formalization allows us to compose an incrementalization of a model analysis from incrementalizations of basic morphisms that reflect the programming language. The correctness of the resulting incrementalization can be proven (cf. Theorem 1) based on the naturality of two transformations—which can be shown for each basic method individually.

From this formalization, we deduced a methodology how implicit incremental computation systems can be made extensible. Thus, they can make use of abstractions incorporated in analysis frameworks also for incremental computation, encouraging modular analyses reusing analysis frameworks (C2). Our approach gives framework developers a tool at hand, which they can use to offer implicit incrementality to their users that is tuned to their framework. For the developer of an analysis, this combines the understandability of a batch specification with the efficiency gained from framework abstractions. This avoids the error-prone process of manual incrementalization and keeps the analysis more readable, thus maintainable.

We have explained the implementation of such an incremental computation system in .NET (C3) and proved its applicability and advantages in terms of implicitly improved performance using a micro-benchmark and a community benchmark (C4). We were able to prove that our approach can provide speedups without additional boilerplate syntax compared to the batch mode versions and is as fast or even faster than other state-of-the-art tools. Our implementation allows the user to choose between batch mode and incremental execution based on a single specification.

10 Future work

Currently, we are applying the functor in a bottom-up fashion to elementary pieces of an abstract syntax tree. The result is a relatively large dynamic dependency graph that makes our approach more memory intensive than other incremental computation techniques, especially if the functions get more complex, but no manual incrementalization is provided. On the other hand, the classical batch mode version of an analysis invoked after every elementary change (or change transaction) can be regarded as a degenerated dependency graph consisting of only a single node where the entire analysis has to be recomputed for every elementary model change. Here, we want to use composition hierarchies such as typically used in model-driven engineering to generalize changes and find granularities in between these two extremes. We hope to coarsen the dynamic dependency graphs with these tech-

niques such that the performance of our approach can be improved.

Implicit techniques always yield the decision whether to apply them or not. In case of incrementality, an incremental approach will always require more memory as intermediate results are saved but on the other hand hopefully give a better performance. The magnitude of performance improvements that can be achieved, if at all, depends on the usage characteristics of the analysis, i.e., what changes are expected and what shares of the intermediate results are typically affected by it. Furthermore, an approach of generalizing changes yields a larger design space than just the binary choice of whether the approach shall be applied or not. However, this design decision does not affect the functional specification of an analysis and is only related to its performance. It can therefore be automated such that given a usage profile, a system could identify pareto-optimal candidates with regard to response time versus memory consumption.

Acknowledgements We would like to thank the anonymous reviewers of the Software & Systems Modeling journal that have helped us to shape the paper into its current form with their thoughtful reviews. This work was partially supported by the MWK (Ministry of Science, Research and the Arts Baden-Württemberg) in the funding line Research Seed Capital (RiSC).

A Details and proofs

In this section, we provide details and proofs for claims made in Sect. 3. In particular, the section is structured as follows: “Appendix A.1” gives a very brief introduction to category theory for reference purposes and to clarify the notation used in the paper. “Appendix A.2” introduces Mutable Type Categories, a representation of mutable object models using category theory. The concept has been originally introduced in the Ph.D. thesis of the first author [39]. “Appendix A.3” discusses the usage of monads for the representation of the incrementalization process. “Appendix A.4” contains the proof of Theorem 1.

A.1 A very brief introduction to category theory

The goal of this section is to briefly introduce most of the category theory that is used in this paper for reference purposes. The concepts are not explained in depth as suitable explanations can be obtained from many textbooks on category theory. For the interested reader, we can recommend the books by Lawvere [75] or Crole [76]. The book by Lawvere is a good general introduction, whereas Crole’s book is more focussed on applications to algebraic type theories.

Definition 3 (Category) A category \mathcal{C} consists of a collection $ob \mathcal{C}$ of objects and collections of morphisms between objects

of \mathcal{C} equipped with an associative operator \circ . Furthermore, for each object A , the identity id_A must exist and for each $f \in Mor(A, B)$ it must hold that $f \circ id_A = f = id_B \circ f$.

For given objects $A, B \in \mathcal{C}$, the set of morphisms is denoted as $Mor_{\mathcal{C}}(A, B)$ or $Mor(A, B)$ if \mathcal{C} is clear from the context. The collection of all morphisms in \mathcal{C} is denoted as Mor with mappings $source, target : Mor \rightarrow ob \mathcal{C}$ determining the source and target object of a morphism.

Remark 3 The associativity means that for any $f \in Mor(A, B), g \in Mor(B, C), h \in Mor(C, D)$ where $A, B, C, D \in \mathcal{C}$ that $(h \circ g) \circ f = h \circ (g \circ f)$.

Example 1 (Sets) One of the most important categories is the category \mathcal{S} of sets. Here, the morphisms are the mappings between sets and the identity for a given set A is the identity mapping on A .

Example 2 (Type systems) As shown by Crole [76], every algebraic type system corresponds to a (Cartesian-closed) category. In the running example, the railway network type system is a category, where the objects are the types such as Semaphore, Switch and Route. The morphisms are the reflexive-transitive closure of the model properties between these types, such as for instance the *position* of a switch, but also combinations such whether the position of the first switch in some collection of switches matches a given (constant) value.

Remark 4 In category theory, equations are often visualized as graphs. Here, objects of a category form the vertices of the graph whereas the directed edges are the morphisms between the objects. The terminology that such a diagram commutes equals to saying that following either path through the diagram yields the same result.

Definition 4 (Functor) A (covariant) functor $\mathcal{F} : \mathcal{C} \rightarrow \mathcal{D}$ between categories \mathcal{C} and \mathcal{D} is a mapping between the objects of \mathcal{C} and \mathcal{D} and the morphisms such that for each objects A and B and $f \in Mor(A, B)$ in \mathcal{C} , we have that $\mathcal{F}(f) \in Mor(\mathcal{F}(A), \mathcal{F}(B))$. Further, a functor has to respect composition, i.e., if $f : A \rightarrow B$ and $g : B \rightarrow C$, then it must hold that $\mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f)$ and $\mathcal{F}(id_A) = id_{\mathcal{F}(A)}$ in \mathcal{D} and for each object A in \mathcal{C} .

Example 3 (Identity functor) An important functor is the identity functor $id_{\mathcal{C}} : \mathcal{C} \rightarrow \mathcal{C}$ for a category \mathcal{C} that maps each object $A \in \mathcal{C}$ to itself and likewise each mapping $\phi \in Mor(A, B)$ to itself.

Example 4 There are three prominent collection functors on \mathcal{S} :

1. The powerset functor $\mathbb{P} : \mathcal{S} \rightarrow \mathcal{S}$ sends each set to its powerset and for each morphism $f : A \rightarrow B$ we have that

$\mathbb{P}(f) : \mathbb{P}(A) \rightarrow \mathbb{P}(B), S \mapsto f(S) := \{f(s) | s \in S\}$.

2. The multiset functor $\mathbb{M} : \mathcal{S} \rightarrow \mathcal{S}$ sends each set S to the set of multisets with elements of S , i.e., to a function $S \rightarrow \mathbb{N}_0$ that assigns each element a multiplicity in the multiset. A morphism $f : A \rightarrow B$ is mapped to

$$\mathbb{M}(f) : \mathbb{M}(A) \rightarrow \mathbb{M}(B), m \mapsto (b \mapsto \sum_{a \in f^{-1}(\{b\})} m(a)).$$

3. The Kleene closure $*$: $\mathcal{S} \rightarrow \mathcal{S}$ maps each set A to its Kleene closure A^* , which formally is the monoid of finite sequences of elements of A . A morphism $f : A \rightarrow B$ is mapped to

$$*(f) : A^* \rightarrow B^*, (a_1; \dots; a_n) \mapsto (f(a_1); \dots; f(a_n)).$$

These three functors can be seen as a formalization of collections.

Example 5 Collections are of course also very present in our running example: As many of the features in the railway network metamodel have a multiplicity higher than one, they refer to collections. A collection of routes, for example, can be typed using the object `Route*`, meanwhile the morphism to get all routes in a railway container is a morphism

$$routes \in Mor(RailwayContainer, Route*).$$

Remark 5 Functors are the ‘natural’ mapping constructs between categories. This is because indeed, the collection of categories forms the category Cat where the morphisms between categories \mathcal{C} and \mathcal{D} (which are themselves objects of Cat) are the functors $\mathcal{F} : \mathcal{C} \rightarrow \mathcal{D}$.

Definition 5 (Natural transformation) A natural transformation $\eta : \mathcal{F} \rightarrow \mathcal{G}$ between two functors $\mathcal{F}, \mathcal{G} : \mathcal{C} \rightarrow \mathcal{D}$ is a set of mappings $\eta_A \in Mor(\mathcal{F}(A), \mathcal{G}(A))$ for each $A \in \mathcal{C}$ (called components of η) such that for each $A, B \in \mathcal{C}$ and $f \in Mor(A, B)$ it holds that $\eta_B \circ \mathcal{F}(f) = \mathcal{G}(f) \circ \eta_A$. That is, the following diagram commutes:

$$\begin{array}{ccc} \mathcal{F}(A) & \xrightarrow{\mathcal{F}(f)} & \mathcal{F}(B) \\ \eta_A \downarrow & & \downarrow \eta_B \\ \mathcal{G}(A) & \xrightarrow{\mathcal{G}(f)} & \mathcal{G}(B) \end{array}$$

If all η_A are isomorphisms, η is called a natural isomorphism between \mathcal{F} and \mathcal{G} .

Example 6 An important example of a natural transformation between functors is the identity transformation on a given functor \mathcal{F} . For each object A in \mathcal{C} , the transformation component for A is the identity, i.e., $(id_{\mathcal{F}})_A = id_{\mathcal{F}(A)}$.

Example 7 A common natural transformation is $first : * \rightarrow id$. For a given type A , this transformation takes a collection of elements in A^* and returns the first element. This transformation is natural, because taking the first element from a collection and processing it further is the same as processing the entire collection and then returning the first element. On the contrary, the operation $first - or - default$ because operations that do not map default values to default values break the commutative diagram from Definition 5.

Definition 6 (Monad) A monad $\mathcal{T} : \mathcal{C} \rightarrow \mathcal{C}$ is a functor equipped with two natural transformations $\eta : id_{\mathcal{C}} \rightarrow \mathcal{T}$ and $\mu : \mathcal{T}^2 \rightarrow \mathcal{T}$ such that $\mu \circ \mathcal{T}\mu = \mu \circ \mu\mathcal{T}$ and $\mu \circ \mathcal{T}\eta = \mu \circ \eta\mathcal{T} = id_{\mathcal{T}}$. Here, the natural transformation η ‘lifts’ objects into the monad and is thus sometimes called the unit operation, while μ simplifies a nested monad.

Example 8 A well-known example of a monad is collections. Here, the functor maps each type A to a generic collection of type A . The functor application of a function corresponds to a mapping. The natural transformation η treats an item of type A as a collection of type A that just contains this element, while μ flattens a collection of collections of type A into a collection of type A . As we already stated that collections are useful for the running example, monad collections are as well.

Definition 7 (Product) Let A and B be objects of a category \mathcal{C} . The product of A and B in \mathcal{C} is an object $A \times B$ of \mathcal{C} together with two projection morphisms $\pi_A : A \times B \rightarrow A$ and $\pi_B : A \times B \rightarrow B$ such that for every object C and every pair of morphisms $f : C \rightarrow A$ and $g : C \rightarrow B$, there is a unique morphism $p : C \rightarrow A \times B$ such that $f = \pi_A \circ p$ and $g = \pi_B \circ p$. That is, the following diagram commutes:

$$\begin{array}{ccccc} & & C & & \\ & f \swarrow & \downarrow p & \searrow g & \\ A & \xleftarrow{\pi_A} & A \times B & \xrightarrow{\pi_B} & B \end{array}$$

Example 9 Products are used for creating tuple types. In the Train Benchmark, tuples are of particular importance because they serve to represent pattern matches.

Definition 8 (Exponential) Let \mathcal{C} be a category such that for each objects A and B their product exists. Then the exponential of A and B is an object A^B together with a morphism $eval : A^B \times B \rightarrow A$ such that for any morphism $f : C \times B \rightarrow A$, there is a unique morphism $\lambda f : C \rightarrow A^B$ such that for every $c \in C$ and $b \in B$, $f(c, b) = eval(\lambda f(c), b)$. That is, the following diagram commutes:

$$\begin{array}{ccc}
C \times B & & \\
\lambda f \times id_B \downarrow & \searrow f & \\
A^B \times B & \xrightarrow{eval} & A
\end{array}$$

Example 10 Exponentials are a formalization of curried functions. They are thus very useful for predicates.

Definition 9 (*Initial object, terminal object*) An initial object \perp of a category \mathcal{C} is an object such that for every object A in \mathcal{C} , there exists exactly one morphism from \perp to A .

Conversely, a terminal object \top of a category \mathcal{C} is an object such that for every object A in \mathcal{C} , there exists exactly one morphism from A to \top .

An initial object of \mathcal{C} is a terminal object of \mathcal{C}^{op} and vice versa. Initial and terminal objects are unique up to isomorphism, i.e., if A and B are initial objects of the same (semi-)category, then there is an isomorphism from A to B .

Example 11 In the category \mathcal{S} of sets, the initial object is the empty set. The terminal objects are the sets that contain exactly one element.

Example 12 In programming, initial and terminal objects usually correspond to the type of `void` and `null`. This is because for any given type, there is exactly one morphism that returns `null` (because there is no other choice) and there is exactly one morphism from `void` to that type (that does not have a specification because the type `void` allows no instances).

Definition 10 A category \mathcal{C} is called Cartesian-closed if it satisfies the following properties:

- It contains an initial and a terminal object.
- For any objects A and B , the product $A \times B$ exists.
- For any objects A and B , the exponential A^B exists.

Example 13 As stated above, Crole has shown that any algebraic type system is isomorphic to a Cartesian-closed category [76]. Thus, we may assume that anything expressible in algebraic type theory can also be expressed in a Cartesian-closed category. This mapping allows us to reason on the program through the corresponding Cartesian-closed category.

A.2 Mutable type categories

The goal of this section is to introduce a formalization of mutable models based on category theory.

The basic idea is to interpret metaclasses as objects of a category, similar to Croles mapping of algebraic type systems to categories [76]. Each metaclass represents the set of possible elements of this class. In contrast to Crole, however,

we consider the mutable state of model elements at runtime. This has a multitude of consequences: First, the value of a member access of an object may be different, depending on the state in which the member was accessed, but the identity of the object is assumed to stay the same. Second, the added complexity in the formalization by Crole to cope with generic methods and functions is not necessary because at runtime, each method is bound already to a type, i.e., we do not have to take open generic type definitions into account. Therefore, we basically rest on the proof made by Crole that any algebraic type system is equivalent to a Cartesian-closed category.

While this foundation on algebraic type theory is useful for many practical applications, we need to keep in mind that model elements have an identity that stays the same even though its attributes or references may change. In particular, models are mutable. This is because according to the general model theory by Stachowiak [77], models always correspond to an original or concept whose identity does not change either.

The goal of considering state is to analyze the impact when this state has changed. Therefore, we are also interested in actions that will change the global states. These operations can be represented as a series of elementary model manipulations.

To account for multiple objects having an interrelated state as, e.g., through opposite references, we model this state as a global state Ω on which we do not make any assumption. This is inspired by the universe Ω commonly used in stochastics. The intuition is that attributes of an object can change over time, just like random variables in stochastics can change over multiple states in the state space.

The reason for a very rough model of a global state is that an elementary model change may change the state not only of the model element that is worked with but also many others. An example here are opposite references where setting a reference of one model element implicitly also sets the opposite reference at another model element. Furthermore, a global state space enables a unified consideration of changes regardless of where the change originates.

The state space Ω can be seen as the space of possible memory states where we abstract from temporary data needed only to compute a given method. Thus, Ω can be thought of the set of sequences over an alphabet (e.g., $\{0, 1\}$) with finitely many nonzero entries. In particular, an element of the function set $\Omega \rightarrow A$ is a typed pointer, very similar to a random variable: For each of the possible memory states (resembling the possible outcomes in probability theory), the value at this memory address may or may not be different.

In the running example, the current position of a switch may change from straight to right or left (which may have an impact on the analysis), but it is still the same switch. In particular, the model element still corresponds to the same

switch in the physical system. In a running application, this switch is typically represented as an object with a memory address in the heap where the current position is carried as an instance field with a fixed offset to the heap address of the switch. In the formalization, this address (heap address of the switch plus offset for the field), can be regarded as a function $\Omega \rightarrow \text{SwitchPosition}$. A memory state ω , however, is the complete state of the entire system.

If a route is extended by a switch position, the switch position automatically references the route it belongs to. If a route is deleted, also all of its switch positions are deleted and references to them are reset. Therefore, a consideration only of the state of the current model element is not sufficient.

One of the merits of category theory is that it often does not require an in-depth understanding of the inner structure of objects but rather reasons on their behavior, i.e., the value or the uniqueness of certain morphisms. This is useful for us, because it enables a formalization at a very high level of abstraction that yields a good flexibility for a later implementation.

To take the global state into account, the basic idea is to extend a static type system (which can be thought of as a category \mathcal{T} through the mapping defined by Crole [76]) with this global state space. The resulting category has as objects the canonical product of objects of $\text{ob } \mathcal{T}$ (i.e., types) and the global state space Ω .

We are particularly interested in the incrementalization of side effect-free morphisms as per the following definition:

Definition 11 (*Side effect-free methods*) The idea of the definition of side effect-free methods is that they do not change the global state. In particular, a function $f : A \times \Omega \rightarrow B \times \Omega$ is side effect free if and only if for all $(a, \omega) \in A \times \Omega$, it holds that

$$\pi_{\Omega}(f(a, \omega)) = \omega$$

where π_{Ω} is the canonical projection to the state of the result.²⁷

Further, another class of methods even ignores the global state entirely, which we capture with the following definition:

Definition 12 (*Stateless Morphism*) Let $f : A \times \Omega \rightarrow B \times \Omega$ be a method. Then f is stateless if and only if it is side effect free and for every $a \in A$ and every $\omega_1, \omega_2 \in \Omega$ we have that

$$\pi_B(f(a, \omega_1)) = \pi_B(f(a, \omega_2)).$$

²⁷ It is common to index projections with indices. However, in the scope of this paper, projections will be indexed with the space they are projecting to, as there is no case of confusion.

Example 14 For any type A , the identity on $A \times \Omega$ is side effect free and stateless.

Proposition 1 A composition of side effect-free morphisms is side effect free. A composition of stateless morphisms is stateless.

Definition 13 (*Mutable Type Category*) A Mutable Type Category (MTC) \mathcal{C} for a set of types $\text{ob } \mathcal{T}$ and a state space Ω is a category that consists of (set-theoretic) tuples $\text{ob } \mathcal{C} := \{A \times \Omega \mid A \in \text{ob } \mathcal{T}\}$ as objects and morphisms $\text{Mor}(A \times \Omega, B \times \Omega)$ between two types A and B as functions $A \times \Omega \rightarrow B \times \Omega$. We further demand that the restriction of \mathcal{C} to side effect-free morphisms \mathcal{C}_{Ω} forms a cartesian-closed category.

Remark 6 If \mathcal{C} is a category, then the restriction \mathcal{C}_{Ω} with $\text{ob } \mathcal{C} = \text{ob } \mathcal{C}_{\Omega}$ and $\text{Mor}_{\mathcal{C}_{\Omega}}(A \times \Omega, B \times \Omega) = \{f \in \text{Mor}_{\mathcal{C}}(A \times \Omega, B \times \Omega) \mid f \text{ is side effect free}\}$ is a category in any case because the composition of side effect-free morphisms is side effect free. Demanding that \mathcal{C}_{Ω} is cartesian-closed means that products and sums exist in \mathcal{C} , for which only side effect-free methods must be taken into account.

Proposition 2 In \mathcal{C}_{Ω} , the product of $A \times \Omega$ and $B \times \Omega$ is $A \times B \times \Omega$ with projections $\pi_A \in \text{Mor}_{\mathcal{C}_{\Omega}}(A \times B \times \Omega, A \times \Omega)$, $(a, b, \omega) \mapsto (a, \omega)$ and $\pi_B \in \text{Mor}_{\mathcal{C}_{\Omega}}(A \times B \times \Omega, B \times \Omega)$, $(a, b, \omega) \mapsto (b, \omega)$.

Proof For methods with side effects, the order in which they are executed matters which makes it hard to reason on product and sum types. If we restrict the methods to side effect-free methods, then the global state is not touched.

In particular, let C be a type and $f \in \text{Mor}_{\mathcal{C}_{\Omega}}(C \times \Omega, A \times \Omega)$, $g \in \text{Mor}_{\mathcal{C}_{\Omega}}(C \times \Omega, B \times \Omega)$. Let $(c, \omega) \in C \times \Omega$. f must map (c, ω) to some $(a_f, \omega) \in A \times \Omega$ because f is side effect free as a morphism in \mathcal{C}_{Ω} . Similarly, g maps (c, ω) to some (b_g, ω) . Then, we define a functor $(f * g) \in \text{Mor}_{\mathcal{C}_{\Omega}}(C \times \Omega, A \times B \times \Omega)$ as follows:

$$(f * g)(c, \omega) = (a_f, b_g, \omega)$$

The projections p_A and p_B obviously satisfy $f = \pi_A \circ (f * g)$ and $g = \pi_B \circ (f * g)$. Furthermore, it is clear that $(f * g)$ is unique.

Hence, $A \times B \times \Omega$ is a product of $A \times \Omega$ and $B \times \Omega$ in \mathcal{C}_{Ω} . \square

As any product of two objects in a category, $A \times B \times \Omega$ is unique up to isomorphism. For example, $B \times A \times \Omega$ would be an alternative.

Remark 7 The object $A \times B \times \Omega$ is not necessarily a product of $A \times \Omega$ and $B \times \Omega$ in \mathcal{C} , basically because it is unclear

how state changes in two morphisms f and g combine. In particular, the model changes performed by f and g may interfere in \mathcal{C} .

Remark 8 The idea is that a mutable type category can be constructed from an algebraic type system with implementations by transforming the type system to the equivalent Cartesian-closed category \mathcal{T} through the mapping defined by Crole. For every object of \mathcal{T} (that represents a type), one creates the set-theoretic product of the set of instances of this type²⁸ and Ω , and obtains \mathcal{C} . Objects of \mathcal{C}_Ω and \mathcal{T} are clearly isomorphic. The morphisms of \mathcal{C} are the morphisms of \mathcal{T} applied to a given state where the effect that the morphism has to the state is determined by its implementation²⁹ and the morphisms of \mathcal{C}_Ω are those that are side effect free. Hence, \mathcal{C}_Ω is Cartesian-closed because \mathcal{T} is.

Definition 14 (Notation) In the remainder of the paper, we use a slightly simplified notation where we write $f : A \rightarrow B$ for $f \in \text{Mor}(A \times \Omega, B \times \Omega)$ when it is clear from context that A and B are types. We also say that $A \in \mathcal{C}$ to denote that $A \times \Omega \in \text{ob } \mathcal{C}$.

Further, a functor \mathcal{F} applied to a given object $A \times \Omega$ in \mathcal{C} must be an object $\mathcal{F}(A \times \Omega) = A' \times \Omega$. We notate this type A' as $A' = \mathcal{F}(A)$ such that $\mathcal{F}(A \times \Omega) = \mathcal{F}(A) \times \Omega$. We refer to changes in the global state as set-theoretic total functions $\Delta\omega \in \Delta\Omega := \Omega \rightarrow \Omega$.

If we know that a morphism $f : A \rightarrow B$ is side effect free, we often treat it as a function $A \times \Omega \rightarrow B$, because it is clear that the state will not change.

Definition 15 In a mutable type category \mathcal{C} , a state change $\Delta\omega$ can be extended to a transformation $Id_{\mathcal{C}} \rightarrow Id_{\mathcal{C}}$

$$\Delta\omega_A : A \rightarrow A, (a, \omega) \mapsto (a, \Delta\omega(\omega))$$

that applies this state change but leaves the value intact. The morphisms ω_A always exist as projections of tuples of the identity on A and the model manipulation. In the remainder, we use this inclusion if this is clear from the context.

A.3 Why not monads?

Although functors already suffice to represent incremental execution systems, it is useful to consider monads. One reason for this is that in order to apply $\mathcal{I}(f)$ to an instance $m \in M$, one needs an instance $m' \in \mathcal{I}(M)$. Since the incrementalization system should be independent of the model

²⁸ We treat a type and its set of instances as two different objects to account for axiomatic set theory, to account for self-descriptive meta-models where some model elements represent their own type [78].

²⁹ Note that a morphism may be implemented by multiple methods according to the mapping of Crole.

type M , such a method should be available as a transformation $Id_{\mathcal{C}} \rightarrow \mathcal{I}$. Semantically, an element of a given type can be regarded as an incremental value that never changes, i.e., as a constant. This definition matches the requirements for the unit transformation η of a monad.

For a given fixed model element, the value for a given property may change over time so that the property value can be understood as an incremental value. A useful thing one would like to achieve is to also apply such a function to incremental values of the model element type and still retrieve an incremental value instead of an incremental value of an incremental value. Such a simplification can be offered by the μ transformation of a monad.

What remains to discuss is whether the naturality of η and μ is useful in this scenario. For η , this naturality means that we could either apply a function on a value and then regard the result as an incremental value (by regarding it as a constant) or lifting the input to the monad (by regarding it as a constant) and then run the incremental derivation of the function on it. This is clearly not the case. Consider, for example, a property access as a function. The value of the property may change over time (if the property is assigned a new value) whereas the constant value obtained by lifting the property access result once does not change. Thus, naturality is something that we explicitly do not want to have for η and we have to be very careful when to apply it.

This situation is different for μ , as this function is only used to combine the incrementality of two levels into one. However, we are typically not interested in *why* the result of a model analysis changed and it suffices to know that the value *has* changed. Therefore, it is viable to lose track of whether the outer or the inner incremental value has caused a value to change.

A.4 Proof of Theorem 1

Proof Let $f : A \rightarrow B$ an arbitrary morphism in \mathcal{C}_Ω and $\Delta\omega \in \Delta\Omega$ be a state change. We begin by proving that (Initialization) holds for f . We first observe that the following diagram commutes due to the naturality of *value*:

$$\begin{array}{ccc} \mathcal{I}(A) & \xrightarrow{\mathcal{I}(f)} & \mathcal{I}(B) \\ \text{value}_A \downarrow & & \downarrow \text{value}_B \\ A & \xrightarrow{f} & B \end{array}$$

We then have that

$$\begin{aligned} & \text{value}_B \circ \mathcal{I}(f) \circ \eta_A \\ &= f \circ \text{value}_A \circ \eta_A \\ &= f \circ (Id_{\mathcal{C}})_A = f. \end{aligned}$$

Here, Id_C denotes the identity functor of C , whose components are the identities, therefore $(Id_C)_A$ is the C -identity on A .

To prove the updates, we see that the following diagram commutes due to the naturality of $apply(\Delta\omega)$:

$$\begin{array}{ccc} \mathcal{I}(A) & \xrightarrow{\mathcal{I}(f)} & \mathcal{I}(B) \\ apply(\Delta\omega)_A \downarrow & & \downarrow apply(\Delta\omega)_B \\ \mathcal{I}(A) & \xrightarrow{\mathcal{I}(f)} & \mathcal{I}(B) \end{array}$$

Thus,

$$\begin{aligned} &value_B \circ apply(\Delta\omega)_B \circ \mathcal{I}(f) \circ \eta_A \\ &= value_B \circ \mathcal{I}(f) \circ apply(\Delta\omega)_A \circ \eta_A \\ &= f \circ value_A \circ apply(\Delta\omega)_A \circ \eta_A \\ &= f \circ value_A \circ \eta_A \circ \Delta\omega_A \\ &= f \circ \Delta\omega_A. \end{aligned}$$

This concludes the proof. \square

References

1. Choi, K., Hwang, S.Y., Blank, T.: Incremental-in-time algorithm for digital simulation. In: Proceedings of the 25th ACM/IEEE Design Automation Conference, pp. 501–505. IEEE Computer Society Press (1988)
2. Salz, A., Horowitz, M.: IRSIM: an incremental MOS switch-level simulator. In: 26th Conference on Design Automation, 1989, pp. 173–178. IEEE (1989)
3. De Lemos, R., Giese, H., Müller, H.A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N.M., Vogel, T., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: Software Engineering for Self-Adaptive Systems II, pp. 1–32. Springer (2013)
4. Szvetits, M., Zdun, U.: Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Softw. Syst. Model.* **15**, 1–39 (2013)
5. Gossman, J.: Introduction to Model/View/ViewModel pattern for building WPF apps (2005). <http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>
6. Smith, J.: PATTERNS-WPF apps with the model-view-viewmodel design pattern. *MSDN Mag.* **24**(2), 72 (2009)
7. Ben-Menachem, M., Marliiss, G.S.: *Software Quality: Producing Practical, Consistent Software*. International Thomson Computer Press, New York (1997)
8. Sutherland, J.: Business objects in corporate information systems. *ACM Comput. Surv. CSUR* **27**(2), 274–276 (1995)
9. Chen, Y., Dunfield, J., Hammer, M.A., Acar, U.A.: Implicit self-adjusting computation for purely functional programs. *J. Funct. Program.* **24**(01), 56–112 (2014)
10. Willis, D., Pearce, D.J., Noble, J.: Caching and incrementalisation in the java query language. *ACM SIGPLAN Not.* **43**(10), 1–18 (2008)
11. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) *Model Driven Engineering Languages and Systems*, pp. 76–90. Springer (2010)
12. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA model transformation system. In: Proceedings of the Third International Workshop on Graph and Model Transformations, pp. 25–32. ACM (2008)
13. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) *Model Driven Engineering Languages and Systems*, pp. 543–557. Springer (2006)
14. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. *Softw. Syst. Model.* **8**(1), 21–43 (2009)
15. Hammer, M.A., Phang, K.Y., Hicks, M., Foster, J.S.: Adapton: Composable, demand-driven incremental computation. *ACM SIGPLAN Not.* **49**, 156–166 (2014)
16. Hammer, M.A., Dunfield, J., Headley, K., Labich, N., Foster, J.S., Hicks, M., Van Horn, D.: Incremental computation with names. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 748–766. ACM (2015)
17. Hinkel, G.: NMF: A Modeling Framework for the .NET Platform, Karlsruhe Institute of Technology, Technical Report (2016)
18. Hinkel, G.: NMF: a multi-platform modeling framework. In: Proceedings of the Theory and Practice of Model Transformations: 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25–29, 2018. Springer (2018) (**accepted, to appear**)
19. Szárnyas, G., Semeráth, O., Ráth, I., Varró, D.: The TTC 2015 train benchmark case for incremental model validation. In: Proceedings of the 8th Transformation Tool Contest, A Part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, L’Aquila, Italy, July 24, 2015, pp. 129–141 (2015)
20. Staron, M.: Adopting model driven software development in industry—a case study at two companies. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) *Model Driven Engineering Languages and Systems*, pp. 57–72. Springer (2006)
21. Mohagheghi, P., Gilani, W., Stefanescu, A., Fernandez, M.A.: An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empir. Softw. Eng.* **18**(1), 89–116 (2013)
22. Meyerovich, L.A., Rabkin, A.S.: Empirical analysis of programming language adoption. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, pp. 1–18. ACM (2013)
23. Carlsson, M.: Monads for incremental computing. *SIGPLAN Not.* **37**(9), 26–35 (2002)
24. Hinkel, G.: Change propagation in an internal model transformation language. In: Proceedings of the Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L’Aquila, Italy, July 20–21, 2015, pp. 3–17. Springer (2015)
25. Hinkel, G., Burger, E.: Change propagation and bidirectionality in internal transformation DSLs. *Softw. Syst. Model* (2017)
26. Hinkel, G.: An NMF solution to the Smart Grid Case at the TTC 2017. In: Proceedings of the 10th Transformation Tool Contest, A Part of the Software Technologies: Applications and Foundations (STAF 2017) Federation of Conferences, series CEUR Workshop Proceedings, CEUR-WS.org (2017)
27. Hinkel, G.: An NMF solution to the Families to Persons case at the TTC 2017. In: Proceedings of the 10th Transformation Tool Contest, A Part of the Software Technologies: Applications and Foundations (STAF 2017) Federation of Conferences, series CEUR Workshop Proceedings, CEUR-WS.org (2017)

28. Wert, A., Happe, J., Happe, L.: Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In: Proceedings of the 2013 International Conference on Software Engineering, series ICSE '13, pp. 552–561. IEEE Press (2013)
29. Tarjan, R.E.: A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.* **18**(2), 110–127 (1979)
30. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. *J. ACM JACM* **22**(2), 215–225 (1975)
31. Holm, J., De Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM JACM* **48**(4), 723–760 (2001)
32. Cattaneo, G., Faruolo, P., Petrillo, U.F., Italiano, G.: Maintaining dynamic minimum spanning trees: an experimental study. *Discrete Appl. Math.* **158**(5), 404–425 (2010)
33. Acar, U.A., Blleloch, G., Ley-Wild, R., Tangwongsan, K., Turkoglu, D.: Traceable data types for self-adjusting computation. *ACM SIGPLAN Not.* **45**, 483–496 (2010)
34. Fraiteur, G.: User-friendly aspects with compile-time imperative semantics in .net: an overview of postsharp. In: Seventh International Conference on Aspect-Oriented Software Development (AOSD) (2008)
35. Fowler, M., Parsons, R.: *Domain Specific Languages*, 1st edn. Addison-Wesley, Reading (2010)
36. Acar, U.A.: Self-adjusting computation. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, USA (2005)
37. Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages. *ACM SIGPLAN Not.* **49**, 145–155 (2014)
38. Grust, T.: *Monad Comprehensions: A Versatile Representation for Queries*. Springer, New York (2004)
39. Hinkel, G.: *Implicit Incremental Model Analyses and Transformations*. Ph.D. thesis, Karlsruhe Institute of Technology (2017) (**to appear**)
40. Hinkel, G.: An NMF solution to the TTC 2018 Social Media Case. In: Proceedings of the 11th Transformation Tool Contest, A Part of the Software Technologies: Applications and Foundations (STAF 2018) Federation of Conferences, series CEUR Workshop Proceedings, CEUR-WS.org (2018)
41. Hinkel, G.: The TTC 2018 Social Media Case. In: Proceedings of the 11th Transformation Tool Contest, A Part of the Software Technologies: Applications and Foundations (STAF 2018) Federation of Conferences, Series CEUR Workshop Proceedings, CEUR-WS.org (2018)
42. Hinkel, G., Happe, L.: An NMF solution to the TTC train benchmark case. In: Proceedings of the 8th Transformation Tool Contest, A Part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, Series CEUR Workshop Proceedings, vol. 1524, CEUR-WS.org, pp. 142–146 (2015)
43. Szárnyas, G., Izsó, B., Ráth, I., Varró, D.: The train benchmark: cross-technology performance evaluation of continuous model queries. *Softw. Syst. Model.* **17**(4), 1365–1393 (2017)
44. Horn, T.: Solving the TTC train benchmark case with funnyqt. In: Proceedings of the 8th Transformation Tool Contest, A Part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, L'Aquila, Italy, July 24, 2015, pp. 147–151 (2015)
45. Wagelaar, D.: The ATL/EMFTVM solution to the train benchmark case. In: Proceedings of the 8th Transformation Tool Contest, A Part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, L'Aquila, Italy, July 24, 2015, pp. 152–156 (2015)
46. Krikava, F.: Solving the ttc'15 train benchmark case study with SIGMA. In: Proceedings of the 8th Transformation Tool Contest, A Part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, L'Aquila, Italy, July 24, 2015, pp. 167–175 (2015)
47. Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: Incquery-d: a distributed incremental model query framework in the cloud. In: Proceedings of the Model-Driven Engineering Languages and Systems—17th International Conference, MODELS 2014, Valencia, Spain, September 28–October 3, 2014, pp. 653–669 (2014)
48. Pugh, W., Teitelbaum, T.: Incremental computation via function caching. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 315–328. ACM (1989)
49. Acar, U.A.: Self-adjusting computation: (an overview). In: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, pp. 1–6. ACM (2009)
50. Hammer, M.A., Acar, U.A., Chen, Y.: CEAL: a C-based Language for self-adjusting computation. *ACM SIGPLAN Not.* **44**, 25–37 (2009)
51. Burckhardt, S., Leijen, D., Sadowski, C., Yi, J., Ball, T.: Two for the price of one: a model for parallel and incremental computation. *ACM SIGPLAN Not.* **46**, 427–444 (2011)
52. Harkes, D., Groenewegen, D.M., Visser, E.: Icedust: Incremental and eventual computation of derived values in persistent object graphs. In: 30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18–22, 2016, Rome, Italy, pp. 11:1–11:26 (2016)
53. Harkes, D., Visser, E.: Icedust 2: derived bidirectional relations and calculation strategy composition. In: 31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19–23, 2017, Barcelona, Spain, pp. 14:1–14:29 (2017)
54. Bainomugisha, E., Carreton, A.L., Cutsem, T.V., Mostinckx, S., Meuter, W.D.: A survey on reactive programming. *ACM Comput. Surv.* **45**(4), 52:1–52:34 (2013)
55. Cooper, G.H., Krishnamurthi, S.: Embedding dynamic dataflow in a call-by-value language. *ESOP* **3924**, 294–308 (2006)
56. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for Ajax applications. *ACM SIGPLAN Not.* **44**, 1–20 (2009)
57. Meijer, E.: Reactive extensions (RX): curing your asynchronous programming blues. In: ACM SIGPLAN Commercial Users of Functional Programming, Series CUFPP '10, ACM, p. 11:1 (2010)
58. Ramalingam, G., Reps, T.: A categorized bibliography on incremental computation. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 502–510. ACM (1993)
59. Reiss, S.P.: An approach to incremental compilation. In: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Series, SIGPLAN '84, pp. 144–156. ACM (1984)
60. Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., Varró, D.: Viatra 3: a reactive model transformation platform. In: Theory and Practice of Model Transformations. Springer, pp. 101–110 (2015)
61. Reps, T.: Optimal-time incremental semantic analysis for syntax-directed editors. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Series, POPL '82, pp. 169–176. ACM (1982)
62. Hoffman, K.: Continuous linq. *Dr. Dobbs J.* **33**(2), 55–58 (2008)
63. Blakeley, J.A., Larson, P.-A., Tompa, F.W.: Efficiently updating materialized views. *SIGMOD Rec.* **15**(2), 61–71 (1986)
64. Gupta, A., Mumick, I.S., et al.: Maintenance of materialized views: problems, techniques, and applications. *IEEE Data Eng. Bull.* **18**(2), 3–18 (1995)
65. Kuno, H.A., Rundensteiner, E.A.: Using object-oriented principles to optimize update propagation to materialized views. In: Proceedings of the Twelfth International Conference on Data Engineering, 1996, pp. 310–317. IEEE (1996)

66. Giarrusso, P.G., Ostermann, K., Eichberg, M., Mitschke, R., Rendel, T., Kästner, C.: Reify your collection queries for modularity and speed! In: Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development, pp. 1–12. ACM (2013)
67. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artif. Intell.* **19**(1), 17–37 (1982)
68. Ujhelyi, Z., Bergmann, G., Hegedűs, Ábel, Horváth, Ákos, Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: an integrated development environment for live model queries, Part 1. *Sci. Comput. Program.* **98**, 80–99 (2015)
69. Ráth, I., Hegedűs, Á., Varró, D.: Derived features for EMF by integrating advanced model queries. In: Vallecillo, A., Tolvanen, J.P., Kindler, E., Störrle, H., Kolovos, D. (eds.) *Modelling foundations and applications*, pp. 102–117. Springer (2012)
70. Bergmann, G.: Translating OCL to graph patterns. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) *Model-Driven Engineering Languages and Systems*, pp. 670–686. Springer (2014)
71. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. *J. Syst. Softw.* **82**(9), 1459–1478 (2009)
72. Reder, A., Egyed, A.: Incremental consistency checking for complex design rules and larger model changes. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) *Model Driven Engineering Languages and Systems*, pp. 202–218. Springer (2012)
73. Uhl, A., Goldschmidt, T., Holzleitner, M.: Using an OCL impact analysis algorithm for view-based textual modelling. *Electron. Commun. EASST* **44**, 1–20 (2011)
74. King, V., Sagert, G.: A fully dynamic algorithm for maintaining the transitive closure. In: Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, pp. 492–498. ACM (1999)
75. Lawvere, F.W., Rosebrugh, R.: *Sets for Mathematics*. Cambridge University Press, Cambridge (2003)
76. Crole, R.L.: *Categories for Types*. Cambridge University Press, Cambridge (1993)
77. Stachowiak, H.: *Allgemeine Modelltheorie*. Springer, New York (1973)
78. Hinkel, G.: Using structural decomposition and refinements for deep modeling of software architectures. *Softw. Syst. Model.* (2018). <https://doi.org/10.1007/s10270-018-0701-6>

Georg Hinkel received his B.Sc. and M.Sc. degrees in Computer Science from the Karlsruhe Institute of Technology (KIT), in 2011 and 2014, respectively, and the B.Sc. degree in mathematics in 2012. In 2017, he received his Ph.D. degree on implicit incremental model analyses and transformations from the KIT. Currently, he is a software technology engineer at Tecan Software Competence Center GmbH. His research interest covers model-driven engineering, incremental-

and medical robotics. He has organized several international work-

shops and is a reviewer for multiple international journals. He is the lead developer of NMF and has (co)-authored more than 30 peer-reviewed publications.

Robert Heinrich is head of the Quality-Driven System Evolution Research Group at Karlsruhe Institute of Technology (Germany). He holds a doctoral degree from Heidelberg University and a degree in Computer Science from University of Applied Sciences Kaiserslautern. His research interests include quality modeling and analysis across several domains, such as information systems, business processes and automated production systems. One core asset of his work is the Palladio software

architecture simulator. He is involved in the organization committees of several international conferences, established and organized various workshops, is reviewer for international premium journals, like IEEE Transactions on Software Engineering and IEEE Software, and is reviewer for international academic funding agencies. Robert is principal investigator or chief coordinator in several grants from governmental funding agencies. He has (co)-authored more than 50 peer-reviewed publications and spent research visits in Chongqing (China) and Tel Aviv (Israel).

Ralf Reussner holds the Chair for Software-Design and -Quality at Karlsruhe Institute of Technology since 2006 and heads the Institute of Program Structures and Data Organisation. His research group works in the interplay of software architecture and predictable software quality as well as on view-based design methods for software-intensive technical systems. Ralf Reussner published over 150 peer-reviewed papers in Journals and Conferences, but also established and organized various

conferences and workshops, including QoSA and WCOP. In addition, he acts as a PC member or reviewer of several conferences and journals, including IEEE Transactions on Software Engineering, IEEE Software and IEEE Computer. He founded the software architecture section of the German Informatics Society in 2006 and is speaker of its software engineering division since 2017. As scientific director of the FZI—Research Center for Information Technologies he consults various industrial partners in the areas of component based software, architectures and software quality. He is principal investigator or chief coordinator in several grants from industrial and governmental funding agencies. Ralf received offers on full professorships from University of Osnabrück, University of Hamburg and Technical University of Munich, which he all rejected. From 2003 till 2006 he held the Juniorprofessorship for Software Engineering at the University of Oldenburg, Germany, and was awarded with a grant of the Emmy-Noether young investigators excellence programme of the National German Science Foundation (DFG).

Repository KITopen

Dies ist ein Postprint/begutachtetes Manuskript.

Empfohlene Zitierung:

Hinkel, G.; Heinrich, R.; Reussner, R.
[An extensible approach to implicit incremental model analyses.](#)
2019. Software and systems modeling, 18.
doi:[10.5445/IR/1000091136](https://doi.org/10.5445/IR/1000091136)

Zitierung der Originalveröffentlichung:

Hinkel, G.; Heinrich, R.; Reussner, R.
[An extensible approach to implicit incremental model analyses.](#)
2019. Software and systems modeling, 18 (5), 3151–3187.
doi:10.1007/s10270-019-00719-y

Lizenzinformationen: [KITopen-Lizenz](#)