Signal Classes: A Mechanism for Building Synchronous and Persistent Signal Networks

Tetsuo Kamina
Oita University, Japan

Tomoyuki Aotani
Mamezou Co.,Ltd., Tokyo, Japan

Hidehiko Masuhara
Tokyo Institute of Technology, Japan

Abstract

Signals are principal abstraction in reactive programming languages and constitute the basics of reactive computations in modern systems, such as the Internet of Things. Signals sometimes utilize past values, which leads to space leak, a problem where accumulated past values waste resources such as the main memory. Persistent signals, an abstraction for time-varying values with their execution histories, provide a generalized and standardized way of space leak management by leaving this management to the database system. However, the current design of persistent signals is very rudimental. For example, they cannot represent complex data structures; they can only be connected using pre-defined API methods that implicitly synchronize the persistent signal network; and they cannot be created dynamically.

In this paper, we show that these problems are derived from more fundamental one: no language mechanism is provided to group related persistent signals. To address this problem, we propose a new language mechanism *signal classes*. A signal class packages a network of related persistent signals that comprises a complex data structure. A signal class defines the scope of synchronization, making it possible to flexibly create persistent signal networks by methods not limited to the use of pre-defined API methods. Furthermore, a signal class can be instantiated, and this instance forms a unit of lifecycle management, which enables the dynamic creation of persistent signals. We formalize signal classes as a small core language where the computation is deliberately defined to interact with the underlying database system using relational algebra. Based on this formalization, we prove the language's glitch freedom. We also formulate its type soundness by introducing an additional check of program well-formedness. This mechanism is implemented as a compiler and a runtime library that is based on a time-series database. The usefulness of the language is demonstrated through the vehicle tracking simulator and viewer case study. We also conducted a performance evaluation that confirms the feasibility of this case study.

2012 ACM Subject Classification Software and its engineering → Object oriented languages; Software and its engineering → Data flow languages; Software and its engineering → Semantics

Keywords and phrases Persistent signals, Reactive programming, Time-series databases

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.17

Related Version *Extended Version*: `https://2020.splashcon.org/details/rebls-2020-papers/5/Managing-Persistent-Signals-using-Signal-Classes`

Funding This research was supported by JSPS KAKENHI Grant Number 17K00115 and 21H03418.

1 Introduction

Signals are principal abstraction in reactive programming languages [10, 17, 30, 33]. Each signal represents a data stream of a periodically updated value. By connecting them, we can declaratively specify dataflow from inputs to outputs. This mechanism was proposed as a representative construct in functional-reactive programming (FRP) [10], and has been available in imperative languages [6, 20, 30, 17, 33]. Signals constitute the basics of reactive computations in modern systems, such as the Internet of Things (IoT).

Persistent signals [18] are the abstraction for time-varying values with their execution histories *in large storage* such as a database. Unlike (transient) signals, past values of persistent signals do not disappear even after the application stops. Furthermore, persistent signals can utilize mostly "unlimited" past values. This means that persistent signals deal with the space leak problem where accumulated past values in transient signals waste resources such as the main memory. Instead, in persistent signals, space leak is managed by the underlying time-series database with large storage. This mechanism is quite useful in applications where a large amount of persistent past values is necessary, such as inspection of accidents in a vehicle tracking system and simulations based on time-series data.

Even though the idea of persistent signals was presented with their implementation and microbenchmarks, the existing persistent signals suffer from the following problems:

- Persistent signals do not provide abstractions for representing complex data structures. For example, when we implement a vehicle tracking system that records x- and y-coordinates of the running vehicle, we need to use two signals corresponding to those coordinates, instead of using one single signal representing "a vehicle." Furthermore, there are no mechanisms for ensuring the simultaneity of updates of those coordinates.
- Persistent signals can be connected only using a predefined set of API methods, which limits the use cases where persistent signals can be applied.
- Persistent signals cannot be created dynamically. This means that we cannot add any new vehicles during the execution of the vehicle tracking system. We consider this limitation to be critical.
- The structures of persistent signal networks are determined statically, and we cannot change them dynamically.

In this paper, we show that all these problems arise from a fundamental restriction: no language mechanism is provided to group and identify the related persistent signals. To address this restriction, we propose *signal classes* for the packaging mechanism of persistent signals. A set of related persistent signals (that form a persistent signal network) is grouped into one signal class. Thus, for example, we can declare "a vehicle" using a signal class where two persistent signals representing its coordinates are declared. Giving an identifier, a signal class can be instantiated dynamically, and this instance defines a scope where all persistent signals are synchronized. This instance also provides a unit of lifecycle management, which makes it easy to provide consistent management regarding lifecycle events, such as dynamic creation and destruction, as well as keeping persistency. Furthermore, by using a persistent signal whose type is also a signal class (this is only an exception to the rule that a persistent signal can have only a "primitive" type, i.e., a type that is supported by the underlying database system), we can dynamically "switch" the persistent signal networks.

We design a programming language that supports signal classes as an extension of SignalJ [17], and show how all the aforementioned problems are tackled using a simple example of vehicle tracking system. Meanwhile, we define an abstract lifecycle model of signal class instances that ensures some properties such as the bindings between persistent signals and database constructs being kept transparent to hide the underlying database from the program; e.g., multiple database tables are not created to store information regarding the same identifier.

The usefulness of the language is demonstrated through the case study of vehicle tracking simulator and viewer. In this application, a vehicle is represented as a signal class instance that encapsulates a persistent signal network comprising the dataflow from the vehicle's coordinates to its velocity. An interactive viewer is implemented using a time-oriented operation representing "scrolling back to a specific time," which is simply realized as a

declarative query on the vehicle. This application scenario implies that the proposed mechanism is useful in applications that handle a time-series data in general, in particular with an interactive user interface that shows both latest and past information. Examples include weather information, IoT sensor monitoring, and SNS timelines.

We also formalize the core language of signal classes. Because a signal class encapsulates its internal time-series data, the language needs to implicitly guarantee its internal consistency. For example, every derived signal at any timestamp must be reproduced from the values of the source signals at that timestamp. In our formalization, the computation is deliberately defined to interact with the underlying database system using relational algebra [5], and based on that, we prove the language's glitch freedom, i.e., a well-typed program does not produce any temporal inconsistencies. We also formulate its type soundness by introducing an additional check of program well-formedness.

We implemented a compiler of signal classes where a signal class is translated into a normal SignalJ class that accesses the runtime library of persistent signals. This runtime library is an extension of the existing persistent signal library [18] where we devise a mechanism that maintains the identities of signal class instances that follow the lifecycle model. This implementation is performed on a time-series database. A performance evaluation is conducted based on this implementation, and its result indicates that the vehicle tracking example is realistic in this implementation.

Contributions of this paper is listed as follows:

- Signal classes that tackle all the aforementioned problems of persistent signals, as well as their instances' lifecycle model, are developed.
- The usefulness of signal classes is demonstrated through the vehicle tracking simulator and viewer case study.
- A core language of signal classes is formalized based on relational algebra (describing the integration between signals and a time-series database) with the proofs of its glitch freedom and type soundness.
- Signal classes are implemented on the basis of a time-series database, which is proven to be responsive through a performance evaluation.

Structure of this paper. Section 2 provides some technical premises on which the proposal of signal classes is based. Section 3 discusses the difficulties in realizing persistent signals. Section 4 introduces signal classes with the descriptions of the aforementioned lifecycle model. The mechanism is explained using the vehicle tracking case study. Section 5 provides the formal definition of the core language with the proofs of glitch-freedom and type soundness. Section 6 illustrates how signal classes are implemented, and shows its performance evaluation results. Section 7 discusses related work, and Section 8 finally concludes this paper.

2 Technical Premises

2.1 Signals

Signals are abstractions for time-varying values that can be declaratively connected to form dataflows. Signals directly represent dataflows from inputs given by the environment to outputs that respond to the changes in the environment. This feature is useful in inplementing modern reactive systems, such as IoT applications. For example, assuming that the power difference of an actuator is calculated by the function `f` that takes a sensor value as an input, both the power difference and the sensor value can be represented as

signals: `powerDifference` and `sensorValue`, respectively. We describe these signals using SignalJ [17], an extension of Java that supports signals.

```
signal int sensorValue = 2000; //initial value
signal int powerDifference = f(sensorValue);
```

These declarations specify that the value of `powerDifference` is recalculated every time the value of `sensorValue` is updated.

Although signals were first proposed in several functional languages such as Fran [10], FrTime [6] and Flapjax [20] introduced their ability to imperatively change signals, and SignalJ also supports this feature using an assignment expression. For example, we can imperatively update the value of `sensorValue`, which is automatically propagated to `powerDifference`.

```
sensorValue = readFromSensors(); //powerDifference is also updated.
```

We note that the dependency between `powerDifference` and `sensorValue` is fixed during the execution, i.e., reassignment of a value to `powerDifference` is not allowed. In SignalJ, imperative update is allowed only for signals that do not depend on other signals, such as `sensorValue`.

SignalJ supports signals with a class type, and does not perceive each internal state change as a distinct update of that signal. Instead, a signal update is perceived only when the identity of the object changes. For example, assuming that the following class `C` declares a signal, namely, `s`, as its field, an assignment to `s` is not considered as a change in the signal of type `C`. We can update that signal by assigning a new instance to that:

```
signal C c = new C("A"); // Class C declares a set of signals.
c.s = 44; // assignment to the internal signal is not propagated to "c"
c = new C("B"); // reassignment to "c" results in the "switched" network
```

This enables SignalJ to perform the switching of signal networks by encapsulating a network of signals as a class instance.

Besides this mechanism to specify the dependency between time-varying values, SignalJ provides specific features that are intensively used throughout this paper. First, in SignalJ, a signal is used anywhere a non-signal is expected. Thus, in the above example, the function `f` can be a method that does not accept a signal but just an integer value. Thus, we can connect signals using legacy library methods that do not support signals, and the dependency between `sensorValue` and `powerDifference` is determined statically. Secondly, in SignalJ, a signal implicitly implements some API methods. One example of such an API method is `subscribe`, which registers an event handler that is called when the receiver signal is updated. In the following section, we will see that query API methods for persistent signals are also provided in this way.

2.2 Persistent signals

One important building block for modern reactive systems is to store time-series data, which are the histories of time-varying values comprising the reactive system. We explain this using the example of a vehicle tracking system [18]. This system records the position of each vehicle, which is obtained from automotive devices. The position changes while the vehicle is moving. In other words, the position of the vehicle is a time-varying value. There are also some other time-varying values that depend on the position, such as the estimated velocity and the total traveled distance of that vehicle. These dependencies on time-varying values motivate us to develop the system using signals. This vehicle tracking system also allows for post analysis (e.g., inspecting the cause of a car accident) and simulation. This means that the change history of each time-varying value stored in the time-series database is necessary.

Table 1 Persistent signal API (selective). In this table, we use T as a type parameter. For example, signal[T] is a type T whose modifier includes `signal`. (Adapted from [18]).

Type	Signature	Description
Basic	`within( java.sql.Timestamp ts, String interval)`	Time-series data within the extent specified by a time-stamp `ts` and `interval` representing its interval
	`bucket(String interval)`	Time-series data using the sampling rate specified by `interval`
Analytic	`avg()`, `max()`, `min()`, etc.	Average, maximum, and minimum value of the receiver signal, resp.
Domain specific	`lastDiff(int i)`	Difference between the current value of the receiver signal and the `i`th value since the last value of that signal
	`distance(signal[T] s)`	Distance between the receiver signal and `s`

Persistent signals [18] are abstractions for time-varying values with their execution histories. A persistent signal is declared as a variant of signals that encapsulates details of its execution history, which is stored in the underlying database. Queries on this execution history are supported by API methods equipped with persistent signals in advance. Each call of the API method is internally translated to the corresponding database query. Because the management of the history is left to the database system, persistent signals solve the space leak problem, where accumulated histories waste resources such as the main memory. Furthermore, persistent signals make their histories available even after the application stops.

In SignalJ, a signal is declared as a persistent signal using the modifier `persistent`. In the following example, `car1234_x` and `car1234_y` are declared as persistent signals whose time-varying values are of type `int`.

```
persistent signal int car1234_x, car1234_y;
signal int c12x = car1234_x.within(Timeseries.now, "12 hours");
signal int c12y = car1234_y.within(Timeseries.now, "12 hours");
```

In this example, these persistent signals represent the position of a specific vehicle; `car1234_x` represents the x-coordinate and `car1234_y` represents the y-coordinate.

Persistent signals are equipped with several query API methods, which are summarized in Table 1. For example, the `within` method shown above returns another persistent signal that contains all the receiver's values that have been recorded within the specified period (the past 12 hours in the above example). In other words, the return value of `within` (`c12x` or `c12y` in the above example) is a *view* of the receiver of `within`. We call such a persistent signal a *view signal*.

View signals are also used to avoid glitches among signals related with the transitive dependency (instead, temporal consistency between signals like `car1234_x` and `car1234_y` must manually be handled by the programmer). SignalJ supports *pull-based* signals, which means that a signal is re-evaluated whenever it is accessed (and it is guaranteed to be glitch-free). This strategy is also applied to the construction of view signals. Each view signal refers to a view that is created by a `SELECT` SQL query corresponding to an API method in Table 1. This query is executed on-demand when the view signal is accessed.

One particular feature of the current implementation of persistent signals is its timing of table and view generation; they are generated at compile time. In other words, lifecycle of persistent and view signals is not considered in the prior work.

2.3 Time-series databases

Time-series databases are used to implement persistent signals. They are usually specialized to store time-series data, and they provide a compact representation and convenient time-oriented API for time-series data. Because time-series data are very common in modern applications, there have been intensive research efforts in this area [16, 8, 26, 1, 19]. There are also many industrial and/or open source implementations of time-series databases, such as TimescaleDB[1], OpenTSDB[2], and InfluxDB[3].

The aforementioned implementation of persistent signals uses TimescaleDB, which is an extension of PostgreSQL, as a backend. Persistent signals have several specific properties: each record has a timestamp; once inserted, entries are not normally updated; and recent entries are more likely to be queried. To effectively interact with such time-series data, TimescaleDB provides an abstraction of a single continuous table across all space and time intervals; this is called a hypertable. All interactions with TimescaleDB (such as SQL queries) are implicitly with hypertables. The preliminary experiments on persistent signals indicate that the existing implementation is sufficiently responsive in most cases.

3 Challenges

The current design of persistent signals suffers from several problems. First, persistent signals do not provide abstractions for representing complex data structures. This problem is indeed illustrated by the aforementioned vehicle tracking example, where the position of the vehicle is represented by two distinct persistent signals, namely, `car1234_x` and `car1234_y`. This is because the persistent signals are only supported with primitive types. Thus, we cannot represent "a vehicle" as one single persistent signal, and the correspondence between x- and y-coordinates and even their synchronization must be manually written by the programmers. This imposes programming with row-level abstractions on the programmers, which is error prone.

Secondly, a view signal must be defined using an API method prepared in advance, where its SQL correspondence is defined. This is because it is difficult to derive a SQL query that creates a view from an arbitrary Java expression. However, this restriction limits the use cases where the persistent signals can be applied. For example, in the vehicle tracking system, we may want to calculate the distance to the destination as follows (assuming that `Position` is a legacy class that is not a part of the persistent signal library):

```
Position target = new Position(..);
signal double dist = target.getDistance(car1234_x,car1234_y);
```

The variable `dist` represents a time-varying value, as its depends on signals `car1234_x` and `car1234_y` (as mentioned above, even though `getDistance` does not expect signals as its arguments, SignalJ can construct a signal network that connects `dist` with `car1234_x` and `car1234_y`). It is also useful if we can use the update history of `dist` derived from database tables for `car1234_x` and `car1234_y`. This is unfortunately difficult because deriving the view is not defined in `getDistance`.

A more serious problem is that persistent signals cannot be created dynamically. This is because the database schema corresponding to persistent signals is determined by the compiler. This approach makes it easy to implement the bindings between persistent signals

[1] `https://www.timescale.com`
[2] `http://opentsdb.net`
[3] `https://www.influxdata.com`

and database constructs because database constructs already exist before the application is running, and their identities do not change during the execution. However, this is a relatively strict limitation. In the vehicle tracking example, this means that every vehicle to be tracked must be statically identified, and we cannot add any vehicles after the application is running. We consider this limitation unacceptable for real applications.

Furthermore, structures of persistent signal networks cannot be changed dynamically. This is because we cannot use a network of persistent signals as a first class citizen. In SignalJ, a signal is always evaluated to the current value (instead of obtaining "the signal itself") when it is accessed. With this semantics, SignalJ supports the switching of signal networks by encapsulating a network of signals as a class instance. However, persistent signals cannot perform this switching because they cannot have a class type.

To understand these problems, we elaborate on the details of them. First, to represent a persistent signal of "a vehicle," we might consider a complex-type persistent signal. For example, Kamina and Aotani noted that existing object-relation mapping might be applied to implement persistent signals with complex types [18], which seems to be straightforward. We define an object (e.g., a vehicle) as a persistent signal, where its internal state changes are considered its execution history maintained by the database table. This might be achieved by defining the mapping from the "`Vehicle`" class to the relation.

Unfortunately, this approach is not as easy as expected. One problem is that there might be deep nesting of internal states where one property of the internal state is another object with a complex type. Furthermore, this contradicts the SignalJ semantics where a signal update is perceived only when the identity of the object changes.

Second, the reason why we want `dist` to be implemented using a view is that we want to update its history simultaneously with updates of `car1234_x` and `car1234_y`. We might consider another approach where every update of `dist` is stored in a separated database table, i.e., `dist` is not a view signal but another persistent signal whose update history is recorded every time `car1234_x` and `car1234_y` are updated. This means that the value of `dist` is no longer calculated by the database query statement creating the view. Instead, the update of `dist` is *pushed*, which is synchronized with pushes of `car1234_x` and `car1234_y`. Thus, we must keep track of which persistent signals are synchronized. For example, there may be a number of vehicles whose updates are independent from others, and we must share the timestamp at the update only among the related persistent signals. The prior work [18] proposes the syntax for parallel assignment of persistent signals to specify the group of synchronous updates; however, intensive use of such a specific syntax makes the program very clumsy.

Finally, the dynamic creation of persistent signals requires the lifecycle of persistent signals to be managed at runtime while supporting persistency. The identity of a dynamically created persistent signal, which is statically unknown, should be retained even after the application crashes. This is because persistent signals are used to ensure tracing of past executions. After the application restarts, this identity should be taken over by the new execution because, for example, it is desirable that the vehicles in the system can be traced again using the records, including those updated before the application stopped. It is also possible that the life of persistent signal ends when the execution history of the corresponding vehicle is no longer necessary.

To enable such lifecycle management, we must keep the consistency between the related persistent signals. For example, in the vehicle tracking system we cannot simply drop `car1234_x` as it is conceptually coupled with another persistent signal `car1234_y`. As the dependencies between persistent signals are implicit in the original work, we must analyze

```
signal class Vehicle {
  String onwer, company, name;
  Position target;

  persistent signal double x, y;
  signal double x12h = x.within(Timeseries.now, "12 hours");
  signal double y12h = y.within(Timeseries.now, "12 hours");
  signal double dx = x12h.lastDiff(1);
  signal double dy = y12h.lastDiff(1);
  signal double v = dx.distance(dy);
  signal double dist = target.getDistance(x,y);

  public Vehicle(String id, String owner, String company,
                 String name, Position target) {
    this.owner = owner; this.company = company;
    this.name = name; this.target = target;
  }
  ...
}
```

Figure 1 Declaration of vehicle using a signal class.

which persistent signal is related to another one, which might be very difficult without assuming hints from the programmers. This leads to the resignation of dynamic persistent signal management.

This observation leads to our hypothesis that all these problems are just instances of more fundamental one: no syntactical support is provided to group and identify the related persistent signals. The lack of grouping mechanism leads to the separate declarations of primitive persistent signals. Such separate declarations are the cause of implicitly constructed persistent signal networks. This implicit construction makes it difficult to identify the set of persistent signals that follow the same lifecycle and can be a unit of switching of persistent signal networks. In the following section, we show that providing this grouping mechanism actually solves all these problems.

4 Signal Classes

To provide a grouping mechanism for persistent signals, we develop a language construct "signal classes" that packages a network of persistent signals into one single class. A signal class itself represents a complex data structure using a set of (primitive) persistent signals. Furthermore, a persistent signal can also have a signal class type, which realizes the dynamic switching of persistent signal networks. By providing an identifier, a signal class can be instantiated dynamically, and the persistent signals enclosed in the signal class are also created when the instance of the signal class is created[4]. Each signal class instance also forms a unit of synchronization and lifecycle management.

[4] More precisely, when the persistent signals are created is determined by the lifecycle model explained in Section 4.2.

An example of a signal class is shown in Figure 1, which declares the "`Vehicle`" class in the vehicle tracking system. A signal class is declared using the modifier `signal` in the class declaration. A persistent signal is declared using the modifier `persistent`, as in the prior work, but now it is declared within a signal class. In Figure 1, two persistent signals, `x` and `y`, are declared to record the position of the vehicle. We note that in this example, we assume that the position of a vehicle, which is monitored by automotive sensors, is periodically sent to a data center that records the vehicle's movement history. Each `Vehicle` instance is an agent reflecting the status of the "real" vehicle identified by the `id` parameter of the constructor. This instance is created at the data center when a new vehicle is registered to the system.

There are also six signals that depend on `x` and `y`, namely, `x12h`, `y12h`, `dx`, `dy`, `v`, and `dist`. While we can imperatively update the values of persistent signals `x` and `y`, any imperative re-assignment for signals depending on `x` and `y` are not allowed, and the values of them are calculated on-demand. Unlike the prior work, the construction of the right-hand side of those signals is not limited to the set of pre-defined API methods. For example, in the right-hand side of the signal `dist`, the receiver `target` of the method `getDistance` is not a signal on which the pre-defined API methods can be called. We assume that the right-hand side of each signal declaration is side-effect-free. For example, we can use the existing checker that checks a method with annotation `@SideEffectFree` does not produce any side-effects[5]:

```
class Position { ...
  @SideEffectFree
  public double getDistance(double x, double y) { ... }
  ... }
```

This check is also necessary to ensure the glitch-freedom (Section 5).

We can still use view signals in this setting. A view signal is a signal whose definition (i.e., the right-hand side of its declaration) is of the form $\texttt{p.m}(\overline{\texttt{e}})$, where `p` is a persistent or view signal, `m` is the name of an API method defined in advance, and each $\texttt{e}_i$ is an argument for `m`. In Figure 1, `x12h`, `y12h`, `dx`, `dy`, and `v` are view signals. We note that the value of view signal is also calculated on demand. One advantage of using view signals is it reduces the update overhead of persistent signals. Furthermore, view signals are useful to "filter" the persistent signals that contain all the execution histories managed by the underlying database system. For example, in Figure 1 we use the `within` query to filter out the old data to avoid performance degradation [18]. Two other API methods, `lastDiff` and `distance` taken from Table 1, are also used.

In summary, the behavior of the `Vehicle` instance is interpreted as follows. Once the instance, namely, `aCar`, of `Vehicle` is created, we can call the `set` method, which is an interface method that all signal classes implicitly implement, to update persistent signals `x` and `y`:

```
aCar.set(33.239148, 131.611722); // setting an initial position.
```

This `set` method first sets the value of `x` and `y` with the provided arguments and then implicitly calculates the value of `dest` using the current values of `x` and `y`. The value of each view signal is automatically determined by the database query statement that creates the corresponding view. For example, `dx` and `dy` calculate the delta between the current value and the last value for each x- and y-coordinate, respectively. The view signal `v` calculates the estimated velocity of the moving vehicle.

[5] `http://checkerframework.org`

4.1 Time-oriented queries on signal class instances

Our vehicle tracking system consists of two subsystems: the vehicle controller (simulating the vehicle's behavior by periodically calling `set`) and the vehicle viewer. Those subsystems run as two distinct processes, and each process has its own signal class instance of the same vehicle. We assume that only the controller continuously updates the vehicles and the viewer accesses each vehicle's time-series data.

To implement the viewer, we can perform time-oriented queries on signal class instances. For example, we can obtain a snapshot of the `aCar` instance at the time when it causes an accident. The following `snapshot` method provides a temporal view of `aCar` at the time specified by the timestamp provided as an argument:

```
aCar.snapshot("2018-06-01T18:10:00").v
```

This query set the internal cursor of the receiver instance to the specified timestamp, which makes the value of every persistent signal on `aCar` be calculated using values of the specified timestamp. This query is called every time the GUI slider is set to point a specific timestamp. Moreover, an argument for `snapshot` can be a variable such as a signal. For example, assuming that a variable `slider` is a signal of "currently selected timestamp using the slider," a vehicle at the time selected by the slider can simply be represented as `aCar.snapshot(slider)`.

We note that the vehicle can still continue to update its persistent signals using `set`, which can be accessed by resetting the cursor to current time using the timestamp "`NOW()`":

```
aCar.snapshot("NOW()");
```

4.2 Lifecycle of a signal class instance

We develop the lifecycle model of signal class instances. In the original work, the underlying database tables for persistent signals are generated by the compiler [18]; this forces persistent signals to be defined statically and makes it very difficult to add new persistent signals at runtime. In the proposed lifecycle model, a signal class instance can be instantiated dynamically, and thus the underlying database tables are generated at runtime. A signal class encapsulates related persistent signals into one module, and this module provides a unit of lifecycle management.

Once created, the history of a signal class instance can exist on the disk even after the application stops. Its identity is preserved on the disk, and when the application restarts, this instance becomes available again from the program. For example, consider the following declaration of a `Vehicle` instance:

```
Vehicle aCar = new Vehicle("501a1234", "Haskell", "Toyota", "Sienta");
```

If there are no database constructs on the disk that correspond to `aCar`, the `Vehicle` instance is created with fresh database constructs. If there already exist such constructs, `aCar` is simply bound to them. In this mechanism, we must keep track of this binding on the disk, and this is done using the `id` parameter, which is mandatory for every constructor in a signal class. This is used as a key to identify the signal class instance.

Figure 2 formalizes this lifecycle model using a state machine diagram. This diagram models one instance that has a full-control to its history (e.g., updating the history like the vehicle controller). There may exist other instances that only perform queries on the history (like the vehicle viewer), but the diagram simply omits such details. Each event that changes its state is triggered by environmental changes or internal program operations. Some of them can be explicitly triggered by calling the interface methods that every signal class implicitly

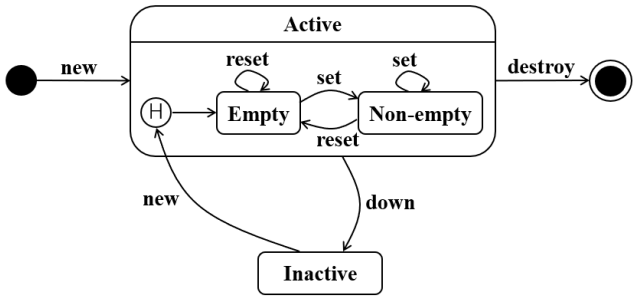

Figure 2 State machine diagram of signal class instance.

```
public interface SignalClassInstance {
  public void set(Object ... newValues);
  public void reset();
  public void destroy();
}
```

Figure 3 Interface for representing signal class lifecycle events.

implements. This interface is shown in Figure 3. We note that this listing imprecisely describes the formal parameters of `set` to indicate that the number of its formal parameters and their types are not defined in advance; the interface of `set` is implicitly derived from the persistent signals declared in that signal class. For example, `set` for `Vehicle` is declared as follows by listing the formal parameters that correspond to the persistent signals `x` and `y`:

```
  public void set(double x, double y);
```

This interface changes by definition of the signal class. The compiler translates the invocation of `set` to make it compatible with the runtime library, which provides the generalized interface.

The states in the lifecycle model are defined as follows:

Initial: The signal class instance has never been created, and there are no persistent or view signals that are bound with this signal class instance.

Active: We can access this signal class instance if it is in this state. A signal class instance will be active just after it is created using the `new` expression. This state consists of the substates **Empty** and **Non-empty**.

Empty: This is the state of a signal class instance where the histories (i.e., the data in the database) of persistent and view signals contained in it are empty. As indicated by the incoming edge from the history state "`H`", **Empty** is the initial substate of the *Active* state, which means that every signal class instance starts with empty histories. The **reset** event also makes the signal class instance empty. Some operations for signals (for example, taking a last value) cannot be performed when the signal class instance is empty.

Non-empty: Persistent and view signals contained in the signal class instance have recorded some of their execution histories. Any operations for signals can be performed when the signal class instance is non-empty.

Inactive: We cannot access this signal class instance if it is in this state. A signal class instance will be inactive if there becomes no pointers that access this instance or the application stops for some reason (e.g., maintenance or crash). A signal class instance preserves its identity on the disk even after it is removed from the main memory.

Final: The signal class instance is destroyed, and persistent and view signals in this instance no longer exist.

The events in the lifecycle model are defined as follows:

new: This event is triggered by the `new` expression, which creates a signal class instance. It generates different side-effects on the signal class instance according to its previous state. If this event occurs with the initial state, it creates new persistent and view signals in the signal class instance with their empty contents. If this event occurs with the inactive state, the signal class instance becomes active and resumes its internal substate, as indicated by the history state "`H`".

set: This event can be generated only when the signal class instance is active. It is triggered when the persistent signals contained in the signal class instance are updated. If there are multiple persistent signals (as in the case of `Vehicle`), their updates are synchronized. The signal class implicitly provides the `set` method for this synchronized update, which makes the state of the signal class instance non-empty.

reset: This event can also be generated only when the signal class instance is active. It is triggered by the `reset` method declared in Figure 3, which cleans the existing histories of the persistent and view signals. This event makes the state of the signal class instance empty.

down: This event is triggered by external or internal environmental changes; it is triggered if the signal class instance can no longer be accessed or the application stops for some reason. After this event, the signal class instance disappears. This instance can however be reactivated, like the "ship of Theseus," using the blueprint of it stored in the database, i.e., when the application restarts, the **new** event can be triggered to restore this instance.

destroy: The signal class instance completes its life when the histories of its persistent and view signals are no longer necessary, and this is performed by generating the destroy event. This event can be fired by calling the `destroy` method shown in Figure 3. It can also be generated by some external environmental changes (e.g., dropping the tables and views from the console of the database management system). After firing this event, no events can be fired on this signal class instance. We note that we can still generate the **new** event using the same identifier (passed to the `id` parameter) again, which starts another independent lifecycle of this `id`.

Importantly, every lifecycle management is performed on the basis of this model. We cannot solely generate, update, or drop the content of each persistent signal. Instead, all related signals are simultaneously generated, updated, and dropped. This makes it easy to ensure data consistency between them.

One important property of the signal class instance is that there should not be multiple signal class instances with the same `id`. This property is ensured according to the lifecycle model. This is because this model does not accept any event sequences where multiple **new** events are triggered until the next **down** or **destroy** events are issued. The **new** event must be the first event of the sequence, and it can follow only the **down** event. Thus, the signal class instance can be activated only when there are no other signal class instances with the same `id`.

4.3 Synchronized update

The vehicle controller continuously updates the histories of the running vehicles. Each signal class instance forms a unit of synchronization. Each signal class provides the `set` method for synchronized update of persistent signals. This improves the synchronized update in the original work [18], where the programmers must ensure that persistent signals that are

defined independently are updated at the same time. For example, we can define the following `run` method in the `Vehicle` class that periodically updates the position of the vehicle:

```
public void run() {
  double[] current = new double[2];
  while (true) {
    current = getGeographicCoordinatesFromSensors();
    set(current[0], current[1]);
  } }
```

This `set` method first computes the value of `dist` using `current[0]` and `current[1]`, and then inserts the triple of current timestamp, `current[0]`, and `current[1]`. Thus, all persistent and view signals are updated at once, and the programmers do not have to worry about any glitches inside the instance.

4.4 Switching network of persistent signals

In SignalJ, we can construct a signal of an object that encapsulates other signals [17]. This feature can be extended to the persistent signals: we can construct a persistent signal of a signal class instance that encapsulates other persistent signals. This allows us to construct a network of persistent signals that changes dynamically, like the "switch" in the FRP languages.

For example, we can construct a signal class that monitors a particular instance of `Vehicle`.

```
signal class Monitor {
  persistent signal Vehicle v;
  public Monitor(String id) { .. } }
```

According to the lifecycle model, we can initialize the persistent signal `v` by issuing the **set** event on the instance `m` of `Monitor`[6]

```
Monitor m = new Monitor("aMonitor");
m.set(aCar);
```

The subsequent **set** events on `m` change the instance of `Vehicle` that `m` monitors, and this change is recorded in the history that is bound with `m`. For example, we may want to monitor some suspicious vehicles more intensively, and the history of `m` is available for inspecting which vehicles were considered suspicious in the past.

We note that SignalJ's object-type signals are considered updated only when the identity of the object changes, and this property is also available in the persistent signals. This means that the persistent signal `v` in `Monitor` does not have to record the instance of `Vehicle` but only its identifier, which is provided by the programmer using the `id` parameter of the signal class constructor.

4.5 Threat to Validity

While the vehicle tracking example well describes the problems of prior work [18] and how signal classes address them, we have not performed any other empirical studies in this work. We consider that discussions in this paper also apply to other applications that handle

[6] We further discuss the initialization issue in Section 6.2.

```
CL  ::=  signal class C { PS FS M }
PS  ::=  persistent signal C p;
FS  ::=  signal C p=e;
 M  ::=  C m(C x) { return e; }
 e  ::=  x | e.p | e.set(e) | e.snapshot(t) | e.m(e) | new C(l) | l | t
 v  ::=  l | t
```

Figure 4 Abstract syntax of signal classes.

timelines of time-varying values that can be identified by some `id`s, such as SNS applications and IoT device monitoring. Further analysis on such application scenarios remain as future work.

5 Formalization

To study important properties of signal classes such as glitch-freedom, we formally define the formal semantics of signal classes based on the simplified syntax of SignalJ shown in Figure 4. The syntax is based on Featherweight Java [15]. Let the metavariables `C` and `D` range over class names; `o` and `p` range over persistent signals; `e` range over expressions; `x` range over variables, which include a special variable `this`; `l` range over identifiers; `t` range over timestamps; `v` range over values; and `m` range over method names. Overlines denote sequences, e.g., $\overline{\texttt{e}}$ represents a possibly empty sequence $\texttt{e}_i, \cdots, \texttt{e}_n$, where n denotes the length of the sequence. We write the length of sequence $\overline{\texttt{e}}$ as $\#(\overline{\texttt{e}})$. We use $\overline{\texttt{C}}\ \overline{\texttt{p}}$ as shorthand for "$\texttt{C}_1\ \texttt{p}_1 \cdots \texttt{C}_n\ \texttt{p}_n$" and $\overline{\texttt{C}}\ \overline{\texttt{s}=\texttt{e}}$ as shorthand for "$\texttt{C}_1\ \texttt{s}_1=\texttt{e}_1 \cdots \texttt{C}_n\ \texttt{s}_n=\texttt{e}_n$."

An expression can be either a variable, an access to a persistent signal, an invocation of special methods `set` and `snapshot` that correspond to the **set** event and a time-oriented query, respectively, a method invocation other than `set` and `snapshot`, an instance creation, or a value that can be either an identifier `l` or a timestamp `t`. The instance creation receives only one identifier as its argument. We assume the set *Id* of identifiers and $\texttt{l} \in \mathit{Id}$. We also assume a total order set *Time* of timestamps where $\bot \in \mathit{Time}$ and $\forall \texttt{t} \in \mathit{Time}.\ \bot \leq \texttt{t}$, i.e., $\bot$ is a bottom element.

In our proposal, there are two kinds of persistent signals: a persistent signal whose value is imperatively set by calling the `set` method, and that whose value is updated on-demand when it is accessed. Figure 4 syntactically distinguishes those two: a persistent signal declaration `PS` representing the former, and a persistent signal declaration `FS` representing the latter; its value is updated on-demand by evaluating the expression `e` that appears in the right-hand side. We note that the latter includes view signals that are calculated using API methods such as database aggregates shown in Section 2.2, as rows in a view are calculated on-demand.

We also apply another simplification to the calculus: the syntax does not provide subclassing, meaning that there are no subtyping rules in the calculus. This is a drastic simplification, but subclassing actually does not interact with the behavior of signal classes, as the execution history is stored for each instance and thus signal lookup is performed in per-instance basis, meaning that the class hierarchy is actually not used during the signal lookup.

A program $(CT, \texttt{e})$ consists of a class table CT that maps a class name `C` to a class declaration `CL` and an expression `e` that corresponds to the body of the `main` method. We

$$\frac{\texttt{signal class C \{ persistent signal } \overline{\texttt{C}}\ \overline{\texttt{p}}\texttt{; ... \}}}{\mathit{sources}(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{p}}}$$

$$\frac{\texttt{signal class C \{ persistent signal } \overline{\texttt{C}}\ \overline{\texttt{p}}\texttt{; ... \}}}{\mathit{signalType}(\texttt{C}, \texttt{p}_i) = \texttt{C}_i}$$

$$\frac{\texttt{signal class C \{ ... signal } \overline{\texttt{C}}\ \overline{\texttt{p}}\texttt{=}\overline{\texttt{e}}\texttt{; ... \}}}{\mathit{signalType}(\texttt{C}, \texttt{p}_i) = \texttt{C}_i}$$

$$\frac{\texttt{signal class C \{ ... signal } \overline{\texttt{C}}\ \overline{\texttt{p}}\texttt{=}\overline{\texttt{e}}\texttt{; ... \}}}{\mathit{signalExpr}(\texttt{C}, \texttt{p}_i) = \texttt{e}_i}$$

$$\frac{\texttt{signal class C \{ ... C}_0\ \texttt{m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{) \{ return e}_0\texttt{; \} \}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}\texttt{.e}}$$

$$\frac{\texttt{signal class C \{ ... C}_0\ \texttt{m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{) \{ return e}_0\texttt{; \} \}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{C}} \to \texttt{C}_0}$$

Figure 5 Auxiliary definitions.

assume that $CT(\texttt{C}) = \texttt{signal class C ...}$ for any $\texttt{C} \in \mathit{dom}(CT)$. We also assume that all signals and methods in the same class, and all parameters in the same method are distinct.

In the following discussion, we use the auxiliary definitions shown in Figure 5. The function $\mathit{sources}(\texttt{C})$ returns a sequence of all pairs of a signal declared with `persistent` and its type in class `C`. The function $\mathit{signalType}(\texttt{p}, \texttt{C})$ returns the type of signal `p` (regardless to say that it is declared with `persistent`) in class `C`. The functions $\mathit{mbody}(\texttt{p}, \texttt{C})$ returns a pair $\overline{\texttt{x}}\texttt{.e}$ of parameters and a method body of method `m` in class `C`. Similarly, $\mathit{mtype}(\texttt{p}, \texttt{C})$ returns a pair $\overline{\texttt{C}} \to \texttt{C}$ of parameter types and a return type of method `m` in class `C`.

5.1 Small-step semantics

We show the reduction rules of expressions in Figure 6. Those are given by the relation of the form $\mu \mid \texttt{e} \longrightarrow \mu' \mid \texttt{e}'$, which is read as "an expression `e` under an environment μ reduces to `e′` under μ'." The environment μ is a set of mapping $\texttt{l} \mapsto \mathcal{R}_{\texttt{C(l)}}$, where `l` is the identifier of the signal class instance, and $\mathcal{R}_{\texttt{C(l)}}$ is an execution history of `l` that is a relation defined as follows:

$$\frac{\mathit{sources}(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{p}}}{\mathcal{R}_{\texttt{C(l)}} = (\mathit{time}, \overline{\texttt{p}})}$$

This relation is handled using the operations provided by the relational algebra [5]: $\pi_{col}(\mathcal{R})$ represents a projection of a relation $\mathcal{R}$ by *col* (i.e., selecting the column *col* from $\mathcal{R}$), and $\sigma_c(\mathcal{R})$ represents filtering $\mathcal{R}$ using the condition c. We often use a singleton set $\{\texttt{l}\}$ and its value `l` interchangeably. We use the predicate *latest*, which is true only if the *time* field of the tuple has the largest value among the relation.

$$\frac{\mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)} \qquad \pi_{\mathtt{p}}(\sigma_{latest}(\mathcal{R}_{\mathtt{C}(\mathtt{l}_0)})) = \mathtt{l}}{\mu \mid \mathtt{l}_0.\mathtt{p} \longrightarrow \mu \mid \mathtt{l}} \quad \text{(R-PSIGNAL)}$$

$$\frac{\mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)} \qquad \mathit{signalExpr}(\mathtt{C}, \mathtt{p}) = \mathtt{e}}{\mu \mid \mathtt{l}_0.\mathtt{p} \longrightarrow \mu \mid \mathtt{e}} \quad \text{(R-VSIGNAL)}$$

$$\mu \mid \mathtt{new\ C(l)} \longrightarrow \mu \oplus (\mathtt{l} \mapsto \emptyset) \mid \mathtt{l} \quad \text{(R-NEW)}$$

$$\frac{\begin{array}{c}\mathcal{R}'_{\mathtt{C}(\mathtt{l})} = \{(\mathtt{t}, \overline{\mathtt{l}})\} \cup \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathtt{t} > \sigma_{latest}(\pi_{time}(\mathcal{R}_{\mathtt{C}(\mathtt{l})})) \\ \mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mu' = \mu \oplus (\mathtt{l} \mapsto \mathcal{R}'_{\mathtt{C}(\mathtt{l})}) \qquad \overline{\mathtt{l}} \in \mathit{dom}(\mu)\end{array}}{\mu \mid \mathtt{l.set}(\overline{\mathtt{l}}) \longrightarrow \mu' \mid \mathtt{l}} \quad \text{(R-SET)}$$

$$\frac{\begin{array}{c}\mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)} \qquad \mathcal{R}'_{\mathtt{C}(\mathtt{l}_0)} = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)} \setminus \sigma_{\mathtt{t} < time}(\mathcal{R}_{\mathtt{C}(\mathtt{l}_0)}) \\ \mu' = \mu \oplus (\mathtt{l}_0 \mapsto \mathcal{R}'_{\mathtt{C}(\mathtt{l}_0)}) \qquad \mathcal{R}'_{\mathtt{C}(\mathtt{l}_0)} \neq \emptyset\end{array}}{\mu \mid \mathtt{l}_0\mathtt{.snapshot(t)} \longrightarrow \mu' \mid \mathtt{l}_0} \quad \text{(R-TIME)}$$

$$\frac{\mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e} \qquad \mathtt{m} \neq \mathtt{set} \qquad \mathtt{m} \neq \mathtt{q}}{\mu \mid \mathtt{l.m}(\overline{\mathtt{l}}) \longrightarrow \mu \mid \mathtt{e}[\overline{\mathtt{x}}/\overline{\mathtt{l}}, \mathtt{this/l}]} \quad \text{(R-INVK)}$$

Figure 6 Small-step computation rules for expressions.

The rules R-PSIGNAL and R-VSIGNAL define how an access to a signal behaves. These rules are straightforward. An access to the persistent signal `p` results in the value in the `p` column of the latest tuple in $\mathcal{R}_{\mathtt{C}(\mathtt{l})}$. An access to a non-source signal results in the right-hand side of its declaration.

The rule R-NEW defines the reduction of the signal class instance creation; it adds the mapping from `l`, which is the identifier of the created instance, to its execution history to μ and returns `l`. We use $\oplus$ as a destructive update of the mapping, i.e., $(x \oplus y)(k) = y(k)$ if k is in the domain of y or $x(k)$ otherwise.

There are three rules for method invocation. In the rule R-SET for the call of `set`, which represents the synchronized update, we first choose a timestamp `t` that is greater than the "largest" value in *Time* and put the tuple $(\mathtt{t}, \overline{\mathtt{l}})$ into the relation. We assume that $\sigma_{latest}(\pi_{time}(\mathcal{R}))$ returns $\bot$ if $\mathcal{R} = \emptyset$. A call of `set` returns the identifier of its receiver, which allows us to describe subsequent computations on the receiver. The rule R-TIME destructively changes the relation $\mu(\mathtt{l}_0)$ so as to be filtered by the given timestamp `t` (as the calculus does not model the database cursor). We note that this is a simplification of the original language's behavior; i.e., unlike the original language, we cannot recover the time-series values that are lost by the `snapshot` operation. R-TIME requires that the resulting relation must not be empty. The rule R-INVK for method invocation (other than `set` and `snapshot`) is straightforward.

We also define the congruence rule that enables a reduction of subexpressions. For this purpose, we first introduce the evaluation context E, which is defined as follows:

$$\begin{array}{rcl} E & ::= & [] \mid E.\mathtt{p} \mid E\mathtt{.set(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{l.set(}\overline{\mathtt{l}}\mathtt{,}E\mathtt{,}\overline{\mathtt{e}}\mathtt{)} \mid E\mathtt{.snapshot(t)} \mid E\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \mid \\ & & \mathtt{l.m(}\overline{\mathtt{l}}\mathtt{,}E\mathtt{,}\overline{\mathtt{e}}\mathtt{)} \end{array}$$

```
signal class C1 {
  persistent signal D p;
  C2 m() { return new C2(l2).set(this); }
}
signal class C2 {
  persistent signal C1 p;
  signal C1 q = this.p.set(new D(l3).set(...));
  signal E n = this.o(this.p.p,this.q.p);
  E o(D x, D y) { ... }
}
```

Figure 7 A glitch-introducing program.

Each evaluation context is an expression with a hole (written []) somewhere inside it. We write $E[\mathsf{e}]$ for an expression obtained by replacing the hole in E with e.

Using E, the congruence rule is defined as follows:

$$\frac{\mu \mid \mathsf{e} \longrightarrow \mu' \mid \mathsf{e}'}{\mu \mid E[\mathsf{e}] \longrightarrow \mu' \mid E[\mathsf{e}']} \qquad \text{(R-Cngl)}$$

The evaluation context syntactically defines the evaluation order of subexpressions in a method invocation, e.g., the arguments are not reduced until the receiver becomes an identifier.

5.2 Static semantics

One significant research question is how the internal state of each signal class instance is kept consistent. This question is also known as an assurance of glitch-freedom. Myter et al. stated that the key intuition behind glitches is that they can only occur for certain topologies of signal networks [23], where two or more propagations from the same source signal (A) join at the other signal (B). A glitch is a situation where the value of B is calculated using A's values with different timestamps.

If no static checking is performed, a glitch can occur even in our simple calculus. Consider the signal classes `C1` and `C2` declared in Figure 7, and the main expression `new C1(l1).m().n`. The signal `n` calls the method `o` that consumes signals `C2.p` and `C2.q`. Those signals depend on the same signal `p` that is a member of `new C1(l1)`. Furthermore, the signal `C2.q` calls the `set` method when its value is accessed. This `set` updates the signal `C1.p` with a new timestamp; thus the signal `n` handles values of the same signal with different timestamps. This is a glitch[7].

Another important role of static analysis is to ensure the calculus type soundness, i.e., to avoid a situation where a program get stuck by, e.g., accessing an undefined attribute in $\mathcal{R}$. To ensure the calculus glitch-freedom and type soundness, we develop a type system of the proposed calculus.

[7] In general, a signal network can contain nodes with different timestamps. Such a network is often useful, as illustrated by the signal network constructed using `lastDiff` (Figure 1, the calculus omits this feature). A glitch is the situation where "the same node" is observed with different timestamps.

$$\frac{\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{C}}}{\Gamma \mid \emptyset \vdash \mathtt{x}_i : \mathtt{C}_i}\ (\text{T-Var}) \qquad \frac{\Sigma = \overline{\mathtt{l}} : \overline{\mathtt{C}}}{\emptyset \mid \Sigma \vdash \mathtt{l}_i : \mathtt{C}_i}\ (\text{T-Id}) \qquad \frac{\mathtt{t} \in \mathit{Time}}{\emptyset \mid \emptyset \vdash \mathtt{t} : \mathtt{T}}\ (\text{T-Ts})$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{signalType}(\mathtt{C}_0, \mathtt{p}) = \mathtt{C}}{\Gamma \mid \Sigma \vdash \mathtt{e}_0\mathtt{.p} : \mathtt{C}}\ (\text{T-Signal})$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{sources}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{p}} \qquad \Gamma \mid \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}}{\Gamma \mid \Sigma \vdash \mathtt{e}_0\mathtt{.set(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{C}_0}\ (\text{T-Set})$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}}{\Gamma \mid \Sigma \vdash \mathtt{e}_0\mathtt{.snapshot(t)} : \mathtt{C}_0}\ (\text{T-Time})$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{C}} \to \mathtt{C} \qquad \Gamma \mid \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}}{\Gamma \mid \Sigma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{C}}\ (\text{T-Invk})$$

$$\Gamma \mid \Sigma, \mathtt{l} : \mathtt{C} \vdash \mathtt{new\ C(l)} : \mathtt{C}\ (\text{T-New})$$

Figure 8 Expression typing.

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \mid \emptyset \vdash \mathtt{e}_0 : \mathtt{C}_0}{\mathtt{C}_0\ \mathtt{m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)}\ \mathtt{\{\ return\ e_0;\ \}\ ok\ in\ C}}\ (\text{T-Method})$$

$$\frac{\mathtt{this} : \mathtt{C} \mid \emptyset \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}} \qquad \mathit{sideeffectfree}(\overline{\mathtt{e}}) \qquad \overline{\mathtt{M}}\ \mathtt{ok\ in\ C}}{\mathtt{signal\ class\ C\ \{\ persistent\ signal}\ \overline{\mathtt{C}}\ \overline{\mathtt{p}}\mathtt{;\ signal}\ \overline{\mathtt{D}}\ \overline{\mathtt{o}}\mathtt{=}\overline{\mathtt{e}}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{\}\ ok}}\ (\text{T-Class})$$

Figure 9 Method and class typing.

Typing rules for expressions are shown in Figure 8. A type environment Γ is a finite mapping from variables to class names. An identifier environment Σ is a finite mapping from identifiers to class names. A type judgment for expressions is of the form $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$, read as "expression `e` is given type `C` under the type environment Γ and identifier environment Σ." To formally describe type judgment, we also introduce a special type `T` for timestamps. As we do not consider any subclasses, there are no subtyping rules in the type system. All typing rules in Figure 8 are straightforward. We note that T-Set checks that the number of arguments for `set` is same as the number of source signals of the receiver, and the type of each argument matches the type of the corresponding persistent signal.

Typing rules for method and class declarations are shown in Figure 9. A type judgment for methods in a class is of the form `M ok in C`, read as "method `M` is well-formed in class `C`." The typing rule T-Method only checks that the method body is given the declared type $\mathtt{C}_0$ under the type environment constructed by formal parameters and the special variable `this`. A signal class `C` is well-formed if the right-hand side expressions $\overline{\mathtt{e}}$ of all the non-source signals are given the declared type under the type environment `this` : `C`, and all methods are well-formed. Furthermore, it checks that each $\mathtt{e}_i$ in $\overline{\mathtt{e}}$ is side-effect-free. As explained

$$\frac{\mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathtt{p} \in \mathcal{R}_{\mathtt{C}(\mathtt{l})}}{\mathit{sources}(\mu, \mathtt{l.p}) = \{\ \mathtt{l.p}\ \}}$$

$$\frac{\mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathit{FP}(\mathtt{C}, \mathtt{p}) = \mathtt{e} \qquad \forall \mathtt{e}_i.\mathtt{p}_i \in \mathtt{e}.\mu \mid \mathtt{e}_i.\mathtt{p}_i \longrightarrow^* \mu_i \mid \mathtt{l}_i.\mathtt{p}_i}{\mathit{sources}(\mu, \mathtt{l.p}) = \bigcup_i \mathit{sources}(\mu_i, \mathtt{l}_i.\mathtt{p}_i)}$$

Figure 10 Source signal lookup.

$$\frac{\mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)} \qquad \mathtt{p}_0 \in \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)}}{\mathit{time}_{\mathtt{l}_0.\mathtt{p}_0}(\mu, \mathtt{l}_0.\mathtt{p}_0) = \pi_{\mathit{time}}(\mathcal{R}_{\mathtt{C}(\mathtt{l}_0)})}$$

$$\frac{\mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathtt{p} \in \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathtt{l}_0 \neq \mathtt{l} \vee \mathtt{p}_0 \neq \mathtt{p}}{\mathit{time}_{\mathtt{l}_0.\mathtt{p}_0}(\mu, \mathtt{l.p}) = \emptyset}$$

$$\frac{\mu(\mathtt{l}) = \mathcal{R}_{\mathtt{C}(\mathtt{l})} \qquad \mathit{FP}(\mathtt{C}, \mathtt{p}) = \mathtt{e} \qquad \forall \mathtt{e}_i.\mathtt{p}_i \in \mathtt{e}.\mu \mid \mathtt{e}_i.\mathtt{p}_i \longrightarrow^* \mu_i \mid \mathtt{l}_i.\mathtt{p}_i}{\mathit{time}_{\mathtt{l}_0.\mathtt{p}_0}(\mu, \mathtt{l.p}) = \bigcup_i \mathit{time}_{\mathtt{l}_0.\mathtt{p}_0}(\mu_i, \mathtt{l}_i.\mathtt{p}_i)}$$

Figure 11 Source signal's time-series.

earlier, our language does not provide this check but relies on an external checker. Thus, we just define the predicate *sideeffectfree*($\mathtt{e}_0$) as follows:

$$\frac{\forall \mathtt{e} \in \text{subexpressions of } \mathtt{e}_0.(\mu \mid \mathtt{e} \longrightarrow^n \mu \mid \mathtt{l} \text{ for some } \mathtt{l} \wedge \forall i \leq n.\mu \mid \mathtt{e} \longrightarrow^i \mu \mid \mathtt{e}' \text{ for some } \mathtt{e}')}{\mathit{sideeffectfree}(\mathtt{e}_0)}$$

This means that each $\mathtt{e}_i$ does not change the runtime environment during its reduction.

5.3 Properties

To formally state the glitch-freedom in our calculus, we further introduce two other auxiliary definitions that perform signal network traverse. Figure 10 defines the source signal lookup $\mathit{sources}(\mu, \mathtt{l.p})$ that returns a set consisting of all the source signals on which `l.p` depend. If `l.p` is a source, which means that the value of `p` is stored in the relation $\mathcal{R}$ of the receiver, $\mathit{sources}(\mu, \mathtt{l.p})$ just returns the singleton of `l.p`. Otherwise, it recursively searches all the source signals by obtaining all signals contained in the right-hand side expression `e` of `l.p`. Similarly, we define the auxiliary definition $\mathit{time}_{\mathtt{l}_0.\mathtt{p}_0}(\mu, \mathtt{l.p})$ that returns the set of timestamps of the source signal $\mathtt{l}_0.\mathtt{p}_0$ *that is observed from signal* `l.p`. Intuitively, the calculus is glitch-free if two or more subexpressions of the right-hand side of `l.p` depends on the same source signal $\mathtt{l}_0.\mathtt{p}_0$, then the same set of timestamps of $\mathtt{l}_0.\mathtt{p}_0$ is observed from any of those subexpressions. We formally describe this property as the following theorem.

▶ **Theorem 5.1** (glitch-freedom). *Let* `signal class C { ... } ok`, $\mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}(\mathtt{l}_0)}$ *for some* μ, *and* $\mathtt{p}_0$ *is a non-source signal declared in* `C` *whose right-hand side is specified as* $\mathtt{e}_0$. *For all subexpressions* `e.p` *and* $\mathtt{e}'.\mathtt{p}'$ *in* $\mathtt{e}_0$, *we have* $\forall s \in \mathit{sources}(\mu, \mathtt{e.p}) \cap \mathit{sources}(\mu, \mathtt{e}'.\mathtt{p}').$ $\mathit{time}_s(\mu, \mathtt{e.p}) = \mathit{time}_s(\mu, \mathtt{e}'.\mathtt{p}')$.

Proof. By `signal class C { ... } ok`, we have *sideeffectfree*($\mathtt{e}_0$). This means that we always access the same μ during the traversals of subexpressions `e.p` and $\mathtt{e}'.\mathtt{p}'$ (i.e., μ does not change in the premises of definitions of *sources*$(\mu, \mathtt{l.p})$ and $time_{\mathtt{l}_0.\mathtt{p}_0}(\mu, \mathtt{l.p})$). Thus, it is obvious that $time_s(\mu, \mathtt{e.p}) = time_s(\mu, \mathtt{e}'.\mathtt{p}')$ for all s in *sources*$(\mu, \mathtt{e.p}) \cap$ *sources*$(\mu, \mathtt{e}'.\mathtt{p}')$. ◀

Another remaining issue is the type soundness. Even though our type system is very simple (e.g., there is no subtyping), formulation of the type soundness is not easy as expected, because our calculus interacts with the database system. We first define the judgment $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$, read "relation $\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$ is well-formed under the environment Σ," which indicates that $\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$ is not empty and all values in $\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$ is well-typed, as follows:

$$\frac{\emptyset \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}_0 \qquad \forall \mathtt{p} \in att(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}).\emptyset \mid \Sigma \vdash \pi_{\mathtt{p}}(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}) : \mathtt{C} \wedge signalType(\mathtt{C}_0, \mathtt{p}) = \mathtt{C} \text{ for some } \mathtt{C}}{\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}}$$

In this definition, we write the set of attributes in $\mathcal{R}$ as $att(\mathcal{R})$. The judgment $\Sigma \vdash \pi_{\mathtt{p}}(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}) : \mathtt{C}$ returns true if all values in $\pi_{\mathtt{p}}(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)})$ have type `C`. Then, we define the well-formedness of a runtime environment as follows.

▶ **Definition 5.2.** *A runtime environment μ is said to be well-formed with respect to an identifier environment Σ, written $\Sigma \vdash \mu$, if $dom(\mu) = dom(\Sigma)$ and $\Sigma \vdash \mu(\mathtt{l})$ for every $\mathtt{l} \in dom(\mu)$.*

We note that the well-formedness of the runtime environment is not always held during the computation. For example, if the redex has the form `new C(l)`, the runtime environment contains an empty relation after the reduction. Thus, the type preservation theorem is formulated as follows.

▶ **Theorem 5.3** (preservasion). *Suppose that $\forall \mathtt{CL} \in dom(CL).\mathtt{CL}$ `ok`. If $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$, $\Sigma \vdash \mu$, and $\mu \mid \mathtt{e} \longrightarrow \mu' \mid \mathtt{e}'$, then $\Gamma \mid \Sigma' \vdash \mathtt{e}' : \mathtt{C}$ for some $\Sigma' \supseteq \Sigma$, and $\mu' = \mu \oplus \{\mathtt{l} \mapsto \emptyset\}$ or $\Sigma' \vdash \mu'$.*

Proof. See Appendix A.1. ◀

We also need to consider the fact that a database query may fail. For example, the type system cannot prohibit the use of a timestamp that is earlier than the beginning of the computation. This observation results in the following progress theorem.

▶ **Theorem 5.4** (progress). *Suppose that $\emptyset \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$ for some `C` and Σ. Then, either `e` is an identifier or a time-oriented query `l.snapshot(t)` where $\mu(\mathtt{l}) = \sigma_{\mathtt{t} < time}(\mu(\mathtt{l}))$ for some μ and `t`, or, for any μ such that $\Sigma \vdash \mu$, there are some expression $\mathtt{e}'$ such that $\mu \mid \mathtt{e} \longrightarrow \mu' \mid \mathtt{e}'$ where $\mu' = \mu \oplus \{\mathtt{l} \mapsto \emptyset\}$ or $\Sigma' \vdash \mu'$ for some $\Sigma' \supseteq \Sigma$.*

Proof. See Appendix A.2. ◀

One issue for ensuring type soundness is that the runtime environment μ can contain an empty relation during the computation. An access to a signal bound with such a relation definitely fails, and to avoid such an access, the empty relation should be populated before an access to the signal occurs. To address this issue, we simply take an approach where all signal class instances are enforced to be immediately populated using `set` after their creations. This is a simplification of the SignalJ's solution discussed in Section 6.2; i.e., the calculus does not model the lifecycle management but is developed to be applicable to this management. To describe this enforcement, we define the well-formedness of expressions.

▶ **Definition 5.5.** *An expression* $\mathtt{e}_0$ *is said to be well-formed (written* $\mathtt{e}$ *wf) if and only if, for all* $\mathtt{e} \in$ *subexpressions of* $\mathtt{e}_0$ *where* $\mathtt{e} = \mathtt{new\ C(l)}$ *for some* $\mathtt{C}$ *and* $\mathtt{l}$, $\mathtt{e}$ *is a subexpression of* $\mathtt{e.set(\overline{e})}$ *for some* $\overline{\mathtt{e}}$ *wf.*

Using this definition, we define the well-formedness of method and class declarations as follows.

$$\frac{\mathtt{e}\ \mathit{wf}}{\mathtt{C\ m(\overline{C}\ \overline{x})\ \{\ return\ e;\ \}}\ \mathit{wf}}$$

$$\frac{\overline{\mathtt{e}}\ \mathit{wf} \qquad \overline{\mathtt{M}}\ \mathit{wf}}{\mathtt{signal\ class\ C\ \{\ ...;\ signal\ \overline{C}\ \overline{p}{=}\overline{e};\ \overline{M}\ \}}\ \mathit{wf}}$$

We write CT *wf* if all class declarations in CT are well-formed. Then, our type soundness theorem is formulated as follows.

▶ **Theorem 5.6** (type soundness)**.** *Consider a program* $(CT, \mathtt{e})$ *with* CT *wf,* $\emptyset \mid \emptyset \vdash \mathtt{e} : \mathtt{C}$*, and* $\mathtt{e}$ *wf. If* $\emptyset \mid \mathtt{e} \longrightarrow^* \mu \mid \mathtt{e}'$ *for some* μ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is either an identifier* $\mathtt{l}$ *with* $\emptyset \mid \Sigma \vdash \mathtt{l} : \mathtt{C}$ *for some* Σ*, or an expression containing a time-oriented query* $\mathtt{l.snapshot(t)}$ *where* $\mu(\mathtt{l}) = \sigma_{\mathtt{t}<time}(\mu(\mathtt{l}))$ *for some* μ *and* $\mathtt{t}$*.*

Proof. By induction on the length of $\emptyset \mid \mathtt{e} \longrightarrow^* \mu \mid \mathtt{e}'$. If $\{\mathtt{l} \mapsto \emptyset\} \in \mu'$ where $\emptyset \mid \mathtt{e} \longrightarrow^* \mu' \mid \mathtt{e}''$ for some $\mathtt{e}''$, it is easy to show that the last applied computation rule is R-New. As $\mathtt{e}$ *wf* and CT *wf*, all `new` expressions in $\mathtt{e}$ and CT are qualified by a `set` call, and because of the evaluation order defined by R-Cngl, the following reduction always use R-Set, resulting in the reduction $\mu' \mid \mathtt{e}'' \longrightarrow_{\text{R-Set}} \mu'' \mid \mathtt{e}'''$ and $\Sigma \vdash \mu''$ for some Σ. Then, Theorems 5.3 and 5.4 finishes the case. Other cases are straightforward. ◀

6 Implementation

The proposed mechanism is implemented on TimescaleDB. We show the runtime architecture of signal class instances in Figure 12. A signal class is compiled into a normal Java class. Each compiled Java class uses the runtime library that prepares the connections to the underlying database system and implements the runtime semantics of persistent signals.

6.1 Compilation

The compiler is implemented using ExtendJ [9]. Figure 12 shows the object diagram after the compilation, where a signal class is translated into a Java class that implements the `SignalClassInstance` interface, which provides the methods necessary for signal class lifecycle management. The implementations of those methods are automatically inserted into the class during the compilation.

Each persistent signal is converted into an instance of `PersistentSignal`, which is a part of the runtime library. Each `PersistentSignal` instance encapsulates the database table that contains all updates of the persistent signal. Every access to the persistent signal is rewritten to the method invocation that returns the "current value" of that signal, and every imperative operation that changes the value of the persistent signal (e.g., a reassignment using `=`) is converted into the method invocation that updates the underlying database table. More precisely, this update is not immediately performed when the reassignment on the persistent signal is issued; it is postponed until the update requests on all the persistent

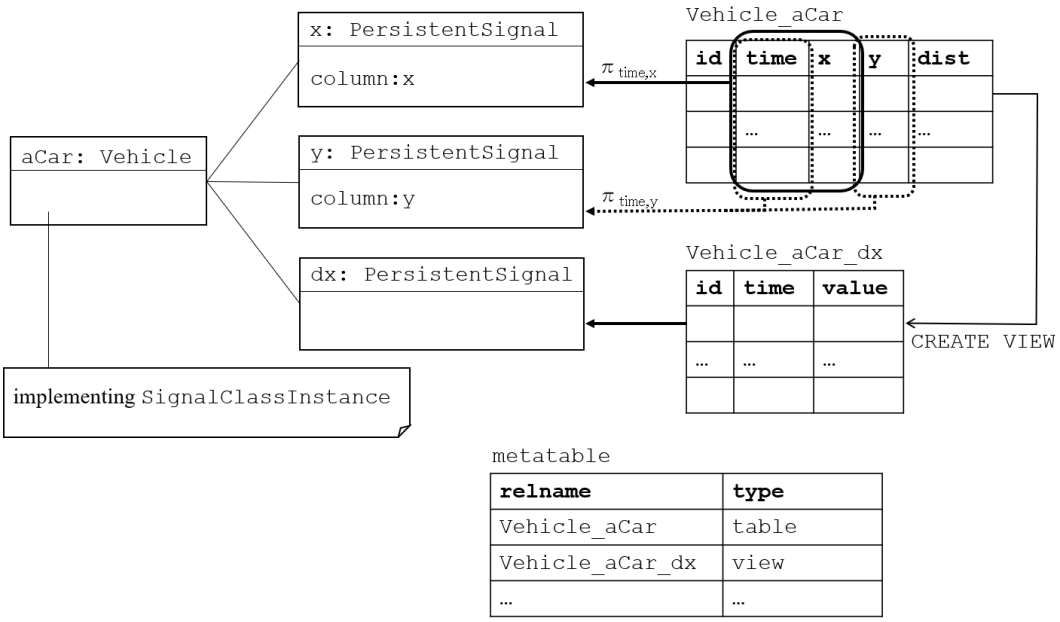

Figure 12 Runtime architecture of signal class instances. Every signal class is introduced with the `SignalClassInstance` interface by the compiler. Each persistent and view signal is converted to an instance of `PersistentSignal`, which is a part of the runtime library that implements the API methods. This instance accesses the projection of the underlying database table and views. There is also the `metatable` that manages the existing signal class instances.

signals contained in the signal class instance are issued. An instance of `Synchronizer`, which is also a part of the runtime library attached to the signal class instance, monitors all the persistent signals in the signal class instance and calls the `set` method that updates the underlying database table according to the provided synchronization policy such as non-blocking-buffered, non-blocking-bufferless, blocking, and asynchronous.

Each constructor of the signal class is also translated to create all `PersistentSignal` instances declared in the signal class. When the instance of `PersistentSignal` is created, it is tested whether the corresponding database table already exists; if so, the `PersistentSignal` instance is connected with that table; if not, a new table is created. Similarly, every view signal, which is also an instance of `PersistentSignal`, is created by calling the API method that is prepared in the runtime library in advance. For the creation of these `PersistentSignal` instances, the compiler simply inserts a piece of code that calls these runtime library methods. As there are chains of dependency between persistent and view signals, these creations of `PersistentSignal` instances are topologically sorted in a similar manner to those in Flapjax [20]. An instance of `Synchronizer` is also created within the constructor execution that monitors all updates of `PersistentSignal` instances.

6.2 Naming and initialization

As explained in Section 4.2, the identifier of a signal class instance is always provided when it is created. In Figure 1, the `id` parameter in the constructor of `Vehicle` is mandatory. Internally, this identifier is used to determine the names of tables and views. The name of persistent signal table is determined by the fully-qualified name of signal class and the identifier. The name of the view is determined by the name of the table and the name of the

view signal. For example, consider the following creation of the `Vehicle` instance again:

```
Vehicle aCar = new Vehicle("501a1234", "Haskell", "Toyota", "Sienta");
```

Assuming that `Vehicle` is declared in the `vehicletracking` package, the names of the table and views are determined as follows.

```
vehicletracking_Vehicle_Oita501a1234      // table for persistent signals
vehicletracking_Vehicle_Oita501a1234_x12h // view for x12h
vehicletracking_Vehicle_Oita501a1234_y12h // view for y12h
vehicletracking_Vehicle_Oita501a1234_dx   // view for dx
vehicletracking_Vehicle_Oita501a1234_dy   // view for dy
vehicletracking_Vehicle_Oita501a1234_v    // view for v
```

The signal class instance encapsulates these table and views. We cannot create multiple `Vehicle` instances with the same identifier, but we can still use the same identifier in the instances of other signal classes.

One subtle issue is providing an initial value to a persistent signal. Theorem 5.6 indicates that the program sticks only when an unexpected timestamp is chosen for the time-oriented query *if the program is well-formed*. This definition of well-formedness requires that a signal class instance should be immediately initialized using `set`. However, in the real program there is also a situation where the instance is bound with an existing database table. In such a case, the call of `set` should not incur any effects.

To ensure that the initialization is performed only when the history of the persistent signal is empty, the `SignalClassInstance` interface in Figure 3 provides an additional method:

```
  public void setIfNotInitialized(Object ... newValues);
```

This method sets the values provided as arguments to persistent signals declared in the receiver signal class instance only if their histories are empty. We note that, like `set`, this interface is defined for the runtime library.

We note that the call of `reset` also makes the execution history empty, and currently our compiler does not check the well-formedness of the program. Instead, the runtime system raises an exception when an empty execution history is accessed.

6.3 Database implementation

TimescaleDB is an open-source time-series database that can run at edge systems as well as in the cloud. Thus, we can implement a variety of applications, including an IoT system where the time-series data is managed in an edge system and a data center that manages massive amount of time-series data. As it is a relational database, the implementation of the dynamic semantics in Section 5, which is based on the relational algebra, is straightforward.

All `PersistentSignal` instances in a signal class instance are connected with the underlying database system when it is created. They access the table for persistent signals and views that corresponds to view signals. Those table and views are created if they do not exist (i.e., if the **new** event is fired with the initial state in Figure 2). The view creating API in Table 1 is also applicable in our system. For example, the expression "`x.within(ts, "12 hours")`", where `x` is a persistent signal containing the x-coordinate of the running vehicle, executes the following `SELECT` query to create a view (`rel_name` is the table that `x` refers to):

```
SELECT time,x FROM [rel_name] WHERE time > ts - interval '12 hours'
```

Table 2 Performance evaluation results. Every measurement is expressed in millisecond (ms), and was performed by taking an average of 10 vehicles.

(a) Response time of persistent/view signal

# records	`x`	`x12h`	`dx`	`v`
100	0.7	0.9	1.2	1.4
1000	0.8	1.0	1.6	1.9
10000	0.8	1.6	6.2	6.1

(b) Overhead of vehicle creation

table/view creation	average time
w/	48.7
w/o	42.1

Normally, the database system does not provide a mechanism to group those related table and views. Thus, the binding between a signal class instance and its corresponding table and views is maintained using the naming mechanism explained in Section 6.2.

Each table for persistent signals of a signal class instance (let the name of this instance be `a`) consists of tuples of persistent signal values with their timestamps. If the persistent signal refers to another signal class instance, the database table contains the name of that instance. When accessed, that instance is restored from the database: if that signal class instance is active, we obtain that instance from the hashtable that contains all active signal class instances; otherwise, a new signal class instance is created using the name stored in the database. The database table `metatable` remembers the names of signal class instances that have been created to date, including inactive ones.

6.4 Performance Evaluation

To confirm that the explained application scenario is realistic in the proposed implementation, we performed simple microbenchmark experiments that measure the response time of persistent signal accesses. These microbenchmarks were performed using TimescaleDB as a backend, which is running on Linux kernel version 4.18.0. This system was running on six-cores Intel Zeon E-2276G 3.80GHz with 16GB main memory and 512GB SSD. TimescaleDB was tuned to have recommended memory settings, including 2GB shared buffers, 6GB effective cache size, 1GB maintenance working memory, and 26,214KB working memory.

In these microbenchmarks, we first prepared histories of vehicles by virtually running them, and then measured the performance of accesses to signals `x` (holding the x-coordinate of each vehicle), `x12h` (holding the last 12 hours of data of x-coordinate), `dx` (holding the difference between `x12h` and its previous value), and `v` (holding the estimated velocity of the vehicle). Before these signals were accessed, each vehicle's timestamp was randomly set by issuing `snapshot`.

Table 2a summarizes the response time of accesses to persistent and view signals. The performance depends on the amount of records the history has. Accesses to view signals `dx` (calculated using join) and `v` (calculated using embedded functions) require around 6 ms when the history has 10,000 records. To confirm that this result is acceptable, we also implemented a vehicle viewer that displays 10 vehicles with 10,000 records and 7 signals (including the y-coordinate `y`, the last 12 hours of y-coordinate `y12h`, and the difference between `y12h` and its previous value, in addition to signals that are shown in Table 2) of each vehicle. This viewer provides a slider to allow the user time-travel, and 70 signals in total are recalculated at once when a specific timestamp is set using the slider. In this viewer, we observed that the slider was mostly responsive.

Table 2b shows the overhead of vehicle creation. This depends on whether the vehicle instance is created by connecting the existing table and views, or creating new ones (i.e., new `id` is introduced). The creation of table and views (it consists of one table and 5 views)

requires around 6 ms. Other overhead includes making connections to the database system. This overhead looks relatively large, but we can reduce this by sharing the connections to the database.

7 Related Work

Signals are a well-known abstraction in reactive programming (RP), which have been inspired by synchronous languages [12, 4, 29] and functional-reactive programming (FRP) languages [10]. FRP features are now available in general-purpose functional languages (e.g., the Yampa library [24] is available for Haskell), and recently they have made their way into imperative object-oriented settings [20, 30, 17] by integrating signals with event-based programming features (such as the event mechanism proposed by EScala [11]).

Even though Yampa's switch and our switching mechanism look somewhat alike, there are fundamental differences between them. First, in our switching, the old sub-network (e.g., the monitored vehicle) is not lost after switching and can be accessed if its `id` is restored. In Yampa, on the other hand, the old signal is lost and we need to preserve every measure manually if we want to access that again. Second, in our switching, there is no guarantee that the switching is performed at the same time when the vehicle is updated, while in Yampa, switches always occur at a global time step. In short, signal classes provide a more general switching with less guarantees.

Although signals in RP languages are not persistent, some research efforts have been made to record the update histories of signals to make them available for debugging. For example, time-traveling [25] makes it possible to pause the execution and rewind to any earlier execution point. This technique is now common in RP debuggers. Reactive Inspector [31], a debugger for REScala [30], visualizes how signal networks are constructed and evolved and how propagations take place over those networks during execution. Using this debugger, a programmer can see the status of the networks at any execution point. Another way of debugging FRP programs is to use *temporal propositions*, an FRP construct based on linear temporal logic [27]. Time-traveling in FRP can also been seen in the literature [28] that presents a uniform way to control how time flows, such as the direction of time flow and sampling rate, by giving time transformations over time domains. Some tools also provide visualization of such time-series data, such as allowing viewing of the execution history in a single display to identify anomaly propagation patterns that are repeated over time [3, 14, 13]. Usually, such tools are dedicated to debugging; thus, they record the history of one execution. Persistence across multiple executions, such as that discussed in the proposed lifecycle model, is not considered. Furthermore, time-series data handled in such tools are not provided for use by applications. For example, no convenient APIs to query over such time-series data are provided.

Other research efforts that are relevant to persistent signals include fault-tolerant RP [21, 22] that provides an implementation for snapshotting mechanism of signals. In contrast, SignalJ focuses more on applications that query over time-series data, which is also evident in the formalization developed using relational algebra. In a larger picture, such time-series data can be open, i.e., that are shared with and queried from multiple processes by referring to that using the identifiers.

As discussed in the implementation of our system, time-series databases provide important techniques to implement persistent signals. Jensen et al. presented a survey on time-series databases, which are also known as time-series management systems [16]. In their survey, time-series databases were categorized as internal data stores, external data stores, and relational

database extensions. An internal data store (e.g., *tsdb* presented by Deri et al. [8]) integrates both a data store and a processing engine together in the same application, allowing for deep integration between the storage and processing engine. In another approach, an external data store (e.g., *Gorilla* by Pelkonen et al. [26] and *BTrDB* by Andersen and Culler [1]) uses an existing external management system, allowing for existing system deployments to be reused. Finally, an relational database extension (e.g., *TimeTravel* by Khalefa et al. [19]) allows the expressive power of the relational database to be applied to the time-series database. TimescaleDB, as used in our implementation, falls into this last category. Overall, there have been many time-series database implementations suitable for different use case scenarios. Therefore, although we consider that the performance of TimescaleDB is satisfactory in many cases, it will be beneficial to consider other implementations that might be suitable for some specific application domain.

Finally, we do not consider the proposed persistent signal lifecyle model as new because there have been much work on persistent objects where the lifetime of the objects can be indefinite (e.g., [2]). We keep the model as simple as possible to extend it to the objects containing a set of time-varying values. There have also been much work on the implementation of persistent objects using SQL (e.g., [7]). Instead, in our system, the mapping is defined only for the trivial cases, i.e., the mapping from persistent signals to the table. We do not define the mapping for view signals; it is left for the programmers or domain engineers. However, we consider some of this definition could be performed automatically using program synthesis. Actually, program synthesis for SQL queries has recently been intensively studied (e.g., the work by [32]). We consider application of such technologies to automatic synthesis of view signals is also an interesting direction for future work.

8 Concluding Remarks

In this paper, we proposed a new language mechanism signal class, which encapsulates a network of related persistent and view signals. Not only does this mechanism allow us to represent persistent time-varying values with complex data types, but it also provides a unit of lifecycle management and a unit of synchronization. All these features overcome the drawbacks of existing persistent signals in that they cannot represent persistent time-varying values with complex data types, they must be created only at compile time, and the network is connected only using pre-defined methods. We clarified how each signal class instance behaves by defining its lifecycle model and formal semantics that maps each signal class instance to the underlying database system using relational algebra. In these definitions, we confirmed several properties regarding database transparency, glitch-freedom, and type soundness. All these results indicate that our approach is effective to implement reactive systems using convenient abstractions of time-varying values with their execution histories.

References

1 Michael P. Andersen and David E. Culler. BTrDB: Optimizing storage system design for timeseries processing. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 39–52, 2016.

2 M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.

3 Herman Banken, Erik Meijer, and Georgious Gousios. Debugging data flows in reactive programs. In *ICSE'18*, pages 752–763, 2018.

4 Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. doi:10.1016/0167-6423(92)90005-V.

5 Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

6 Gregory H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Department of Computer Science, Brown University, 2008.

7 S. Dar, N.H̃. Gehani, and H.Ṽ. Jagadish. CQL++: A SQL for the Ode object-oriented DBMS. In *Advances in Database Technology — EDBT '92*, volume 580 of *LNCS*, pages 201–216, 1992.

8 Luca Deri, Simone Mainardi, and Francesco Fusco. tsdb: A compressed database for time series. In *TMA*, 2012.

9 Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'07)*, pages 1–18, 2007. doi:10.1145/1297105.1297029.

10 Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 263–273, 1997. doi:10.1145/258949.258973.

11 Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: modular event-driven object interactions in Scala. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD'11)*, pages 227–240, 2011. doi:10.1145/1960275.1960303.

12 Nicholas Halbwachs, Paul Caspi, Pascal Paymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. doi:10.1109/5.97300.

13 Takumi Hikosaka, Tetsuo Kamina, and Katsuhisa Maruyama. Visualizing reactive execution history using propagation traces. In *REBLS'18*, 2018.

14 Jeff Horemans and Bob Reynders. Elmsvuur: A multi-tier version of elm and its time-traveling debugger. In *TFP 2017*, volume 10788 of *LNCS*, pages 79–97, 2017.

15 Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

16 Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 29:2581–2600, 2018.

17 Tetsuo Kamina and Tomoyuki Aotani. Harmonizing signals and events with a lightweight extension to Java. *The Art, Science, and Engineering of Programming*, 2(3), 2018. doi:10.22152/programming-journal.org/2018/2/5.

18 Tetsuo Kamina and Tomoyuki Aotani. An approach for persistent time-varying values. In *Onward!'19*, pages 17–31, 2019.

19 Mohamed E. Khalefa, Ulrike Fischer, Torben Bach Pedersen, and Wolfgang Lehner. Model-based integration of past & future in TimeTravel. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 1974–1977, 2012.

20 Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA'09)*, pages 1–20, 2009. doi:10.1145/1640089.1640091.

21 Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant distributed reactive programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:26, 2018.

22 Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.*, 3(OOPSLA):144:1–144:29, 2019.

23 Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Distributed reactive programming for reactive distributed systems. *The Art, Science, and Engineering of Programming*, 3(3):5:1–5:52, 2019.

24 Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell'02)*, pages 51–64, 2002. doi:10.1145/581690.581695.

25 Laszlo Pandy. Bret Victor style reactive debugging. Elm Workshop, 2013.

26 Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, 2015.

27 Ivan Perez. Back to the future: time travel in FRP. In *Haskell'17*, pages 105–116, 2017.

28 Ivan Perez and Henrik Nilsson. Testing and debugging functional reactive programming. *Proceedings of the ACM on Programming Languages*, 1, 2017.

29 Marc Pouzet. *Lucid Synchrone version 3.0: Tutorial and Reference Manual*. Université Paris-Sud, LRI, April 2006. Online manual. URL: https://www.di.ens.fr/~pouzet/lucid-synchrone/lucid-synchrone-3.0-manual.pdf.

30 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY'14)*, pages 25–36, 2014. doi:10.1145/2577080.2577083.

31 Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In *ICSE'16*, pages 796–807, 2016.

32 Sai Zhang and Yuyin Sun. Automatically synthesizing SQL queries from input-output examples. In *ASE'13*, pages 224–234, 2013.

33 YungYu Zhuang. A lightweight push-pull mechanism for implicitly using signals in imperative programming. *Journal of Computer Languages*, 54, 2019.

A Proofs

A.1 Proof of Theorem 5.3

We first show some lemmas required by the proof of Theorem 5.3.

▶ **Lemma A.1** (weakening).
1. *If* $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{x} \notin \Gamma$, *then* $\Gamma, \mathtt{x} : \mathtt{D} \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$.
2. *If* $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{l} \notin \Sigma$, *then* $\Gamma \mid \Sigma, \mathtt{l} : \mathtt{C} \vdash \mathtt{e} : \mathtt{C}$.

Proof. By straightforward induction on $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$. ◀

▶ **Lemma A.2** (substitution). *If* $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{C}} \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0$ *and* $\Gamma \mid \Sigma \vdash \overline{\mathtt{l}} : \overline{\mathtt{C}}$, *then* $\Gamma \mid \Sigma \vdash [\overline{\mathtt{l}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{C}_0$.

Proof. By induction on $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$. ◀

Proof of Theorem 5.3. By induction on the derivation of $\mu \mid \mathtt{e} \longrightarrow \mu' \mid \mathtt{e}'$.

Case R-PSIGNAL: $\mathtt{e} = \mathtt{l}_0\mathtt{.p} \quad \mathtt{e}' = \mathtt{l} \quad \mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)} \quad \pi_{\mathtt{p}}(\sigma_{latest}(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)})) = \mathtt{l}$

By T-SIGNAL, $\Gamma \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}'_0$ and $signalType(\mathtt{C}'_0, \mathtt{p}) = \mathtt{C}$ for some $\mathtt{C}'_0$. As $\Sigma \vdash \mu$, we have $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$, and by the definition of $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$, we have $\emptyset \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}_0$. By Lemma A.1, $\Gamma \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}_0$. Thus $\mathtt{C}'_0 = \mathtt{C}_0$. By the definition of $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$ and T-SIGNAL, we have $\pi_{\mathtt{p}}(\sigma_{latest}(\mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}))$. Then, Lemma A.1 finishes the case.

Case R-VSIGNAL: $\mathtt{e} = \mathtt{l}_0\mathtt{.p} \quad \mathtt{e}' = \mathtt{e}_0 \quad \mu(\mathtt{l}_0) = \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)} \quad signalExpr(\mathtt{C}_0, \mathtt{p}) = \mathtt{e}_0$

By T-SIGNAL, $\Gamma \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}'_0$ and $\mathit{signalType}(\mathtt{C}'_0, \mathtt{p}) = \mathtt{C}$ for some $\mathtt{C}'_0$. As $\Sigma \vdash \mu$, we have $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$, and by the definition of $\Sigma \vdash \mathcal{R}_{\mathtt{C}_0(\mathtt{l}_0)}$, we have $\emptyset \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}_0$. By Lemma A.1, $\Gamma \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}_0$. Thus $\mathtt{C}'_0 = \mathtt{C}_0$. By T-CLASS and the definitions of *signalExpr* and *signalType*, we have $\mathtt{this} : \mathtt{C}_0 \mid \emptyset \vdash \mathtt{e}_0 : \mathtt{C}$. Then, Lemma A.1 finishes the case.

Case R-NEW: $\mathtt{e} = \mathtt{new\ C(l)} \quad \mathtt{e}' = \mathtt{l} \quad \mu' = \mu \oplus \{\mathtt{l} \mapsto \emptyset\}$

Let $\Sigma' = \Sigma, \mathtt{l} : \mathtt{C}$. By T-ID, $\Sigma' \vdash \mathtt{l} : \mathtt{C}$, finishing the case.

Case R-SET: $\mathtt{e} = \mathtt{l.set}(\overline{\mathtt{l}}) \quad \mathtt{e}' = \mathtt{l}$

It is easy to show that $\Gamma \mid \Sigma \vdash \mu'$, and by T-SET we have $\Gamma \mid \Sigma \vdash \mathtt{l} : \mathtt{C}$, finishing the case.

Case R-TIME: $\mathtt{e} = \mathtt{l}_0\mathtt{.snapshot(t)} \quad \mathtt{e}' = \mathtt{l}_0$

It is easy to show that $\Gamma \mid \Sigma \vdash \mu'$, and by T-TIME we have $\Gamma \mid \Sigma \vdash \mathtt{l}_0 : \mathtt{C}$, finishing the case.

Case R-INVK: Finished by Lemma A.2, T-INVK, and definitions of *mtype* and *mbody*.

Case R-CNGL: Finished by the induction hypothesis. ◀

A.2 Proof of Theorem 5.4

Proof of Theorem 5.4. By induction on the derivation of $\Gamma \mid \Sigma \vdash \mathtt{e} : \mathtt{C}$.

Cases T-VAR and T-TS: Cannot occur.

Case T-ID: Immediately finished.

Case T-SIGNAL: $\mathtt{e} = \mathtt{e}_0\mathtt{.p} \quad \emptyset \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{signalType}(\mathtt{C}_0, \mathtt{p}) = \mathtt{C}$

There are three subcases based on the form of $\mathtt{e}_0$:

Subcase 1: $\mathtt{e}_0 = \mathtt{l}_0$

There are further subcases based on the definition of *signalType*: (1) `signal class C {... persistent signal` $\overline{\mathtt{C}}$ $\overline{\mathtt{p}}$`; ...}` and $\mathtt{p} \in \overline{\mathtt{p}}$. Assuming $\emptyset \mid \Sigma \vdash \mu$, we have $\Sigma \vdash \mu(\mathtt{l}_0)$, i.e., we have a non-empty $\mu(\mathtt{l}_0)$. Thus, $\pi_{\mathtt{p}}(\sigma_{\mathit{latest}}(\mu(\mathtt{l}_0)) = \mathtt{l}$ for some $\mathtt{l}$. Thus, R-PSIGNAL can be applied to $\mathtt{e}$, finishing the case; (2) `signal class C {... signal` $\overline{\mathtt{C}}$ $\overline{\mathtt{p}}$`=`$\overline{\mathtt{e}}$`; ...}` and $\mathtt{p} \in \overline{\mathtt{p}}$. Similarly, R-VSIGNAL finishes the case.

Subcase 2: $\mathtt{e}_0 = \mathtt{l}_0\mathtt{.snapshot(t)}$

Immediately finished because this is the case where $\mathtt{e}$ is an expression containing a time-oriented query.

Subcase 3: Otherwise, R-CONG finishes the case.

Case T-SET: $\mathtt{e} = \mathtt{e}_0\mathtt{.set}(\overline{\mathtt{e}}) \quad \emptyset \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C} \quad \emptyset \mid \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$

There are subcases based on the form of $\mathtt{e}_0$:

Subcase 1: $\mathtt{e}_0 = \mathtt{l}_0$

There are further subcases based on the form of $\overline{\mathtt{e}}$: (1) $\overline{\mathtt{e}} = \overline{\mathtt{l}}$. By T-ID, $\overline{\mathtt{l}} \in \mathit{dom}(\Sigma)$, and assuming $\Sigma \vdash \mu$, we have $\overline{\mathtt{l}} \in \mathit{dom}(\mu)$. We can choose $\mathtt{t} \in \mathit{Time}$ such that $\mathtt{t} > \sigma_{\mathit{latest}}(\pi_{\mathit{time}}(\mu(\mathtt{l}_0)))$. Let $\mathcal{R}'_{\mathtt{C}(\mathtt{l}_0)} = \{(\mathtt{t}, \overline{\mathtt{l}})\} \cup \mu(\mathtt{l}_0)$ and $\mu' = \mu \oplus (\mathtt{l}_0 \mapsto \mathcal{R}'_{\mathtt{C}(\mathtt{l}_0)})$. Then, R-SET finishes the case; (2) Otherwise, R-CONG finishes the case.

Subcase 2: Otherwise, R-CONG finishes the case.

Case T-TIME: $\mathtt{e} = \mathtt{e}_0\mathtt{.snapshot(t)} \quad \emptyset \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C} \quad \emptyset \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$

There are subcases based on the form of $\mathtt{e}_0$:

Subcase 1: $\mathtt{e}_0 = \mathtt{l}_0$

Immediately finished because this is the case where $\mathtt{e}$ is an expression containing a time-oriented query.

Subcase 2: Otherwise, R-CONG finishes the case.

Case T-INVK: $\mathtt{e} = \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}}) \quad \emptyset \mid \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{C}} \to \mathtt{C} \quad \emptyset \mid \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}}$

There are subcases based on the form of $\mathtt{e}_0$:

Subcase 1: $\mathtt{e}_0 = \mathtt{l}_0$

There are further subcases based on the form of $\overline{\mathtt{e}}$: (1) $\overline{\mathtt{e}} = \overline{\mathtt{l}}$. By the definition of *mtype* and *mbody*, we have $mbody(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{x}}.\mathtt{e}$ where the number of $\overline{\mathtt{l}}$ and that of $\overline{\mathtt{x}}$ are the same. Thus, R-Invk finishes the case; (2) Otherwise, R-Cong finishes the case.

Subcase 2: Otherwise, R-Cong finishes the case.

Case T-New: Immediately finished. ◀