Managing Persistent Signals using Signal Classes

A Work-In-Progress Paper

Tetsuo Kamina
Oita University
Japan
kamina@acm.org

Tomoyuki Aotani
Mamezou Co.,Ltd.
Japan
tomoyuki-aotani@mamezou.com

Hidehiko Masuhara
Tokyo Institute of Technology
Japan
masuhara@acm.org

Abstract

Persistent signals provide a convenient abstraction for time-varying values with their execution histories by implicitly leaving the management of execution histories to the database system. The current design of persistent signals is very rudimental. For example, they do not provide abstractions for representing complex data structures, and they can only be connected using API methods prepared in advance. To make matters worse, dynamic creation of persistent signals is not allowed. In this paper, we show that these problems can be addressed by introducing a new language mechanism called *signal classes*. A signal class packages a network of related persistent signals that comprises a complex data structure. A signal class can be instantiated dynamically, and this instance is considered a "small world," where all signals within it are synchronous. We further show that this synchronous world makes it easy to realize persistent signal networks connected by expressions that are not limited to the API methods. This dynamic creation of signal class instances is managed by a simple lifecycle model where both dynamic lifecycle events and persistency are defined.

1 Introduction

Signals offer a convenient abstraction for time-varying values. Each signal can represent a data stream with a periodically updated value, and by connecting them, we can declaratively specify a dataflow from inputs to outputs. This mechanism became a representative construct in functional-reactive programming (FRP) [Elliott and Hudak 1997], and now it is common in imperative languages [Kamina and Aotani 2018; Salvaneschi et al. 2014; Zhuang 2019]. Signals constitute the basics of reactive computations in modern systems, such as the Internet of Things (IoT). Such systems also diligently use past values. Even though signals may utilize past values, doing so is likely discouraged because, e.g., using arbitrary far past value leads to "time leak", a problem where accumulated past values waste resources such as the main memory. Thus, most programming languages supporting signals practically limit the usage of past values.

Persistent signals [Kamina and Aotani 2019] are abstractions for time-varying values with their execution histories. They provide the generalized and standardized way of time leak management by leaving this management to the time-series database system. Furthermore, the execution histories do not disappear even after the application stops. The details of the underlying database system are completely hidden from the program.

Even though this idea was presented with their implementation and microbenchmarks, the existing persistent signals suffer from the following problems. First, persistent signals do not provide abstractions for representing complex data structures. For example, the paper on persistent signals describes a vehicle tracking system as an example, where persistent signals are used to record x- and y-coordinates of the running vehicle. Because persistent signals only support primitive types, it is necessary to represent each coordinate using an individual persistent signal, and the correspondence between those coordinates (e.g., their simultaneous updates) must be explicitly maintained by the programmer. Second, persistent signals can be connected only using a set of API methods, where the mapping from queries on persistent signals to the corresponding database queries are prepared in advance. This limits the use cases where persistent signals can be applied. Finally, persistent signals cannot be created dynamically, as the database schema that implements the persistent signals is implicitly derived from the source code. This means that we cannot add any new vehicles during the execution of the vehicle tracking system. We consider this limitation to be critical. Additionally, we cannot change the structures of persistent signal networks dynamically.

Why is the design of persistent signals so disorderly? We identified the following reasons. First, mapping from persistent signal networks to the underlying database system is a non-trivial issue. For example, at first we may consider that the existing object-relation mapping can be applied to implement persistent signals with complex data structures. However, this approach is not consistent with the existing object-based signals, where internal state changes within an object are not observed as an update of the signal value. Next, we must avoid glitches (temporal inconsistencies within the signal networks) in not only the current values but also all past values. For this purpose, persistent signals record the *logical time* for all past updates, which are shared with all the synchronized persistent signals. Thus, we must keep track of which persistent signals are synchronized, which makes

REBLS'20, Nov 16, 2020, Virtual
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the program clumsy. One compromise is to abandon this keeping track by limiting this synchronization only for the "views" defined by the prepared API methods. Finally, this abandonment of keeping track of related persistent signals also makes it difficult to keep the data consistency between persistent signals that are dynamically created.

In this paper, we propose *signal classes* to address all these problems. First, a signal class provides a packaging mechanism for persistent signals. This means that we can construct a complex data structure in the form of a persistent signal network, while restricting every persistent signal to have only a type that is supported by the underlying database system (and thus we need not worry about the mapping between complex data structures). One exception is that a signal class can be nested, i.e., within a signal class, we can declare a persistent signal whose type is also a signal class. This is a simple extension of object-type signals in SignalJ [Kamina and Aotani 2018] that allows dynamic switching of signal networks. Secondly, a signal class can be instantiated dynamically. This dynamically instantiated instance is considered a small world, where all persistent signals enclosed in that instance are synchronized. Thus, this abstraction simply provides a means of synchronization management for related persistent signals without introducing clumsy code, and now we can connect persistent signals using expressions not limited by the prepared API methods. 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.

2 Technical Premises

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 [Kamina and Aotani 2018], 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 [Cooper 2008; Elliott and Hudak 1997; Meyerovich et al. 2009], in SignalJ, we can imperatively change the value of signals. For example, we can imperatively update the value of `sensorValue`, which is automatically propagated to `powerDifference`.

```
\\the value of powerDifference is also updated.
sensorValue = readFromSensors();
```

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`.

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 value is expected, by implicitly unlifting the signal to a normal Java-value (i.e., the latest value of the signal). 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 of `subscribe` is updated. In the following section, we will see that query API methods for persistent signals are also provided in this way.

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 [Kamina and Aotani 2019]. 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.

Persistent signals [Kamina and Aotani 2019] 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, we can apply a generalized way to manage the "time leak",

where accumulated histories waste resources such as the main memory, using larger storage. 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, the variables `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. 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 (this means that consistency between signals like `car1234_x` and `car1234_y` must explicitly 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; e.g., if a view signal depends on multiple persistent signals, a join is performed among them to keep the signal network consistent.

One particular feature of the current implementation of persistent signals is its timing of table and view generation; they are generated at compile time. This is because it is considered that the database schema should be available before the application is running.

Persistent signals are implemented using TimescaleDB[1], a time-series database that is an extension of PostgreSQL. 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 where time-oriented queries (e.g., aggregation based on time intervals) are performed.

3 Challenges

Persistent signals were presented as a preliminary research proposal with their proof-of-concept implementation. Although there are microbenchmark results indicating that this research direction is promising, 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 position of a vehicle" as one single persistent signal, and the correspondence between x- and y-coordinates and even their synchronization must be maintained 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 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

[1] https://www.timescale.com

```
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

any vehicles after the application is running. We consider this limitation unacceptable for real applications.

4 Signal Classes

We consider that the origin of the aforementioned problems is a lack of abstraction to identify the related persistent signals that follow the same lifecycle. To address those problems, we propose a new language construct, which we call "signal classes", that packages a network of persistent signals into one single class. We consider the source of the problems being that each persistent signal is independently declared. For example, if we can package related persistent signals into one class, the update history of a complex data structure can be represented using a set of those persistent signals while keeping the types of them limited to primitive types supported by the underlying database system. An instance of signal class and its enclosing persistent signals follow the lifecycle model explained later.

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, and it must declare persistent signals as its members. In Figure 1, two persistent signals, `x` and `y`, are declared to record the position of the vehicle. There are also six signals that depend on `x` and `y`, namely, `x12h`, `y12h`, `dx`, `dy`, `v`, and `dist`. These signals are categorized as view signals and persistent signals.

A view signal is a signal whose definition (i.e., the right-hand side of its declaration) is of the form $\mathtt{p.m(\overline{e})}$, where `p` is a persistent or view signal, `m` is the name of an API method defined in advance, and each $\mathtt{e}_i$ is an argument for `m`. In Figure 1, `x12h`, `y12h`, `dx`, `dy`, and `v` are view signals. The value of view signal is calculated *on demand* using the source persistent signals.

On the other hand, the signal `dist` is not a view signal, as the receiver `target` of the method `getDistance` is neither a persistent nor a view signal. In this case, we cannot calculate a view for `dist`, and the update history of `dist` is directly stored on the disk. The update of `dist` is synchronized with the updates of `x` and `y`. Thus, the value of `dist` always reflects the current values of `x` and `y`. The boundary of this synchronization is determined by the scope of the signal class instance, i.e., all persistent signals in the same signal class instance are synchronized.

This synchronized update makes it possible to connect persistent signals using not only API methods prepared in advance but also other expressions that connect persistent and view signals. However, view signals are still useful in our system. One advantage of using view signals is it reduces the update overhead of persistent signals. Another advantage is that they lead to better database design, as views are usually derived from the table and having values like `dist` in the table does not meet the "third normal form" property of relational databases.

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`:

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

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.

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. This means that each signal class instance encapsulates the network of persistent and view signals that comprises a vehicle and is considered a unit of lifecycle, synchronization, and database management. As explained below, this idea addresses all the problems mentioned in the previous section.

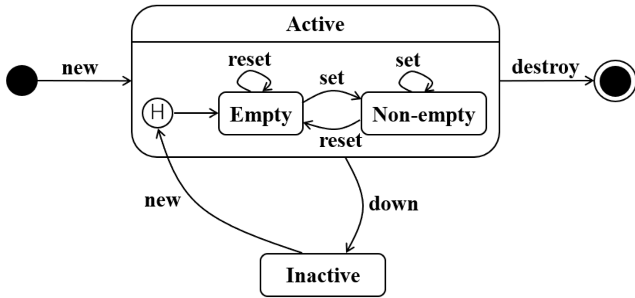

Figure 2. State machine diagram of signal class instance

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

Figure 3. Interface for representing signal class lifecycle events

Lifecycle of a signal class instance. We first explain the lifecycle model of signal class instances. In the original work, the underlying database tables for persistent signals are generated by the compiler [Kamina and Aotani 2019]; 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, 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, i.e., the programmer should choose the key when instantiating a `Vehicle`.

Figure 2 formalizes this lifecycle model using a state machine diagram. 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 implements. This interface is shown in Figure 3. We note that this interface is defined for the SignalJ runtime library, which is written in Java. In the source code, 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 the 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. These interface methods throw an exception when their calls break the lifecycle contract shown in Figure 2.

Other events are not triggered by these interface methods. The **new** event is triggered by the `new` expression, which creates a signal class instance. The **down** 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.

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 necessary because the signal class instance and its persistent signals encapsulate the underlying database. The access to the database is transparent. Thus, we must prohibit the side-effect where one update from one of the signal class instances affects the results from the other side[2].

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`.

There can be situations where the accessed persistent and view signals are empty. For example, if the view accesses the old data that is missing, this view simply becomes empty, and we might access the signal with the empty view. In such

[2]One exception is that we may still manage the histories of persistent signals using the console of the database system.

a case, a runtime exception is thrown to notify that this signal is not ready. This situation can definitely occur as the lifecycle starts with the empty signals.

Synchronized update. 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 [Kamina and Aotani 2019], 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 = getGeoCoordinatesFromSensors();
    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[0]`, `current[1]`, and `dist` with the current timestamp. Thus, all persistent and view signals are updated at once, and the programmers do not have to worry about any glitches inside the instance.

Switching network of persistent signals. In SignalJ, we can construct a signal of an object that encapsulates other signals [Kamina and Aotani 2018]. 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 [Nilsson et al. 2002][3].

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`.

```
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.

[3]Precisely, our switching differs from the Yampa's switch in significant ways. This difference is discussed in Section 5.

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.

Implementation. Signal classes are implemented as an extension of SignalJ. Currently this is a *prototype* that is a subject to change, but will be publicly available by merging it with the original SignalJ in the future. The implementation details are left out due to the page limit.

5 Related Work

Signals are a well-known abstraction in reactive programming (RP), which have been inspired by synchronous languages [Berry and Gonthier 1992; Halbwachs et al. 1991; Pouzet 2006] and functional-reactive programming (FRP) languages [Elliott and Hudak 1997]. FRP features are now available in general-purpose functional languages (e.g., the Yampa library [Nilsson et al. 2002] is available for Haskell), and recently they have made their way into imperative object-oriented settings [Kamina and Aotani 2018; Meyerovich et al. 2009; Salvaneschi et al. 2014] by integrating signals with event-based programming features, such as the event mechanism proposed for EScala [Gasiunas et al. 2011].

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 [Pandy 2013] makes it possible to pause the execution and rewind to any earlier execution point. This technique is now common in RP debuggers. Reactive Inspector [Salvaneschi and Mezini 2016], a debugger for REScala [Salvaneschi et al. 2014], 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. In other words, these tools stores time-series values for each signal to make time-traveling possible. Another way of debugging FRP programs is to use *temporal propositions*, an FRP construct based on linear temporal

logic [Perez 2017]. Time-traveling in FRP can also been seen in the literature [Perez and Nilsson 2017] 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. Time-traveling debuggers 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.

6 Concluding Remarks

We have described that not only do signal classes allow us to represent time-varying values with complex data types, but they also provide a unit of lifecycle management and synchronization. All these features overcome the drawbacks of existing persistent signals. We clarified how each signal class instance behaves by defining its lifecycle model. We consider that our approach is promising to implement reactive systems using convenient abstractions of time-varying values with their execution histories, i.e., "time leak" is not a problem but a convenient building block to implement modern reactive systems.

Signal classes are still work-in-progress and we are planning to proceed to obtain more rigorous research results. First, as our mechanism involves synchronous objects that are however connected with each other and thus there may be dataflows between them. Thus, an assurance of glitch-freedom is an issue. For this purpose, we are designing a formal calculus of signal classes to prove glitch-freedom. We are also implementing signal classes over the cloud-based time-series database (by setting up TimescaleDB on the cloud) to develop more realistic applications based on signal classes.

References

Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

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

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*. 263–273. https://doi.org/10.1145/258949.258973

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

Nicholas Halbwachs, Paul Caspi, Pascal Paymond, and Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language Lustre. *Proc. IEEE* 79, 9 (1991), 1305–1320. https://doi.org/10.1109/5.97300

Tetsuo Kamina and Tomoyuki Aotani. 2018. Harmonizing Signals and Events with a Lightweight Extension to Java. *The Art, Science, and Engineering of Programming* 2, 3 (2018). https://doi.org/10.22152/programming-journal.org/2018/2/5

Tetsuo Kamina and Tomoyuki Aotani. 2019. An Approach for Persistent Time-Varying Values. In *Onward!'19*. 17–31.

Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. 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)*. 1–20. https://doi.org/10.1145/1640089.1640091

Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell'02)*. 51–64. https://doi.org/10.1145/581690.581695

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

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

Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proceedings of the ACM on Programming Languages* 1 (2017).

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

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

Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *ICSE'16*. 796–807.

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