IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 27, 2020, accepted June 16, 2020, date of publication July 20, 2020, date of current version August 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3010697

A Semantic Framework for the Design of Distributed Reactive Real-Time Languages and Applications

MATEO SANABRIA-ARDILA[1], LUIS DANIEL BENAVIDES NAVARRO[1], (Member, IEEE), DANIEL DÍAZ-LÓPEZ[2], AND WILMER GARZÓN-ALFONSO[1]

[1]Escuela Colombiana de Ingeniería Julio Garavito, Bogotá 111166, Colombia
[2]School of Engineering, Science and Technology, Universidad del Rosario, Bogotá 111711, Colombia

Corresponding author: Luis Daniel Benavides Navarro (luis.benavides@escuelaing.edu.co)

This work was supported in part by the Escuela Colombiana de Ingeniería Julio Garavito through the Project Diseño y Construcción de Herramientas Reactivas con Aplicaciones a Middleware Distribuido Para el Procesamiento de Grandes Volumenes de Datos, and in part by the Department of Applied Mathematics and Computer Science, Universidad del Rosario.

ABSTRACT The proliferation of on-demand internet services delivered over a network of a heterogeneous set of computing devices has created the need for high-performing dynamic systems in real-time. Services such as audio and video streaming, self-driving cars, the Internet of things (IoT), or instant communication on social networks have forced system designers to rethink the architectures and tools for implementing computer systems. Reactive programming has been advocated as a programming paradigm suitable for implementing dynamic applications with complex and heterogeneous architectural needs. However, there is no consensus on the core set of features that a reactive framework must-have. Furthermore, the current set of features proposed in reactive tools seems very restricted to cope with the actual needs for concurrency and distribution in modern systems. In this paper, several alternative semantics for distributed reactive languages are investigated, addressing complex open issues such as glitch avoidance, explicit distribution support, and constructs for explicit time management. First, we propose a reactive event-based programming language with explicit support for distribution, concurrency, and explicit time manipulation (ReactiveXD). Second, we present a reactive event-based semantic framework called Distributed Reactive Rewriting Framework (DRRF). The framework uses rewriting logic to model the components of a distributed base application, observables, and observers, and predicates supporting explicit time manipulation. Finally, to validate the proposal, the paper discusses the specification of the semantics of ReactiveXD and a scenario describing a case of intrusion detection on IoT networks.

INDEX TERMS Distributed computing, the Internet of Things (IoT), logical clocks, Maude, real-time languages, reactive programming, rewriting logic, cybersecurity applications.

I. INTRODUCTION

The ubiquity of internet access and the proliferation of mobile devices, mesh networks, and IoT appliances have changed the architecture of applications and digital services. These applications are now decentralized, concurrent, and highly-interactive, responding in real-time to local and remote stimulus in an ever-changing environment. Consider, for example, the problem of detecting security attacks in an IoT ecosystems composed by hundreds of heterogeneous components spread over different places. Such a scenario may integrate different kinds of sensors, actuators, and computing devices exchanging messages concurrently and in real-time over a dynamic distributed network. Protecting this system is a challenging task because the configuration of the system is dynamic, the system has to adapt itself in real-time to physical changes, and current mainstream tools do not provide the abstractions to cope with these requirements. Existing approaches have tried to address security on IoT networks by means of reactive patterns, but they use restricted forms of reactivity, e.g., detection of sequences of events, or guarded sequences. Thus, an IoT scenario or any other featured as decentralized, concurrent, interactive, and based on events, determines the need to define clever mechanisms able to:

The associate editor coordinating the review of this manuscript and approving it for publication was Javier Medina.

i) support reactive and dynamic applications, ii) provide time management functionalities and iii) be integrated into scenarios composed by heterogeneous and distributed elements.

In this context, reactive programming has been advocated as a programming paradigm suitable for the implementation of dynamic applications with complex and heterogeneous architectural needs [1], and proponents defend it as a ''better'' alternative to other programming paradigms like Object Oriented Programming, actor languages, functional programming, among others.

However, there is no consensus on the core set of features that a reactive framework must have. The original proposal of functional reactive programming [9] seems very different from its modern counterparts inspired by ReactiveX[1]: RxAda [21], RAY [13]), MPML3D [23] or LUSTRE [12]. Furthermore, despite of distribution and concurrency being first class concerns in modern applications, there is very little research on semantics and implementation of fully distributed reactive frameworks and its usefulness for the development of these decentralized, concurrent, and highly interactive applications.

In this paper, several alternative semantics for distributed reactive languages are investigated, addressing complex open issues such as glitch avoidance, explicit distribution support, and constructs for explicit time management. Particularly the investigation considers the applicability of distributed reactive languages in the context of IoT scenarios showing how different semantics may influence programming patterns and programming languages. Concretely, this paper presents the following contributions:

- An updated state of the art of reactive programming languages and frameworks that improves previous reviews (e.g., [1]) through the analysis of the implications and implementations when explicit support for distribution and time management is considered.
- A novel reactive oriented programming language, ReactiveXD, with explicit support for distribution and time management constructors.
- A configurable and executable semantic framework (DRRF), developed in MAUDE [2] using rewriting logic. This framework contains a minimal set of building blocks for the implementation of distributed real-time reactive languages. DRRF is used to study the semantic proposed for ReactiveXD.
- Finally, the paper shows the implementation of an IoT scenario using the distributed reactive framework (DRRF) configured with ReactiveXD's semantics.

This document is organized as follows. Section II presents an overview of functional reactive programming and reactive programming à la ReactiveX. Then, section III introduces ReactiveXD, a reactive language with explicit support for distribution and time management. Section IV presents DRRF, a semantic framework implemented in MAUDE using rewriting logic. Section V presents an evaluation of the applicability of DRRF, showing how to model the semantics of ReactiveXD, and showing the implementation of an IoT scenario and how to prove properties on it. Section VI discusses related work. Finally, conclusions and future works are presented in sections VII and VIII.

[1] http://reactivex.io/

[2] http://maude.cs.illinois.edu/

II. REACTIVE PROGRAMMING OVERVIEW

In this section, reactive programming is introduced through a discussion of functional reactive programming as initially proposed, and reactive programming à la ReactiveX as an example of current practitioners' approach to reactive programming[1].

A. FUNCTIONAL REACTIVE PROGRAMMING

Initially motivated by interactive 3D computer graphics, functional reactive programming (FRP) [9] proposes as main abstraction values that vary continuously over time. Those values, called *behaviors*, were intended to describe the continuous change over time of graphical objects in an animation. *Behaviors* may also depend on other time-varying entities, creating in this way dynamic dependent objects. Thus, whenever a *behavior* is updated, all *behaviors* depending on it are also updated, e.g., like formulas depending on cells in a spreadsheet. The following code shows a simple declaration of two *behaviors*:

```
varTime = timeBehavior();
varTimex10 = varTime ∗ 10;
```

The function `timeBehavior()` returns a *behavior* that is a time-varying entity whose value varies **continuously**. The `varTimex10` variable also represents a *behavior* that is updated when `varTime` is updated and contains ten times the value of `varTime`. Note that to build a dynamic application, the programmer does not need to update the values explicitly, the values are updated automatically and the language runtime should take care of the complexity of updating values asynchronously. Note that *behaviors* vary continuously over time, i.e., they are not considered discrete entities, on the contrary, their change is modeled as a continuous function over time. In this case, time is also a continuous changing entity.

Functional reactive languages also provide an abstraction called *events*. *Events* refer to occurrences in the real world, e.g., mouse clicks, or key presses. From this, *behaviors* may be defined in terms of reactions to *events*, but they will still have declarative semantics in the function of time. For example, a valid *event* can be the first message arriving after time `t0` and may be expressed as:

```
firstMessageAfter(t0)
```

and a *behavior* depending on this *event* may be:

```
changingBehavior =
varTime untilB firstMessageAfter(t0) ∗=>
    varTimex10
```

This *behavior* will return the same value as *behavior* `varTime` until the event `firstMessageAfter(t0)` when it starts returning the same value as *behavior* `varTimex10`. `untilB` is a native infix construct introduced in the original FRP paper [9] and returns the *behavior* to the left until the *event* to the right appears, afterward it starts returning the *behavior* associated with such *event*. The `*=>` event handler returns a behavior depending only on time.

The implementation of functional reactive programming (FRP) languages has several considerations like evaluation model, lifting, glitch, and multi-directionality. The first consideration is the *evaluation model*, which is affected by how changes propagate in the dependency graph. The *evaluation model* may be *pull-based* (when the propagation is triggered by value requests) or *push based* (when the propagation is driven by *behavior* updates). *Lifting* is another consideration that exists when a functional reactive language is embedded in another programming language. In such a case, some mechanism must be put in place to make primitive operators and custom functions to operate on behaviors. *Glitch* avoidance is another consideration. For example, in the propagation of new values during the program execution, a naive implementation may lead to update inconsistencies when a computation is executed before all its *behaviors* get updated, leading to a *glitch*. Finally, it is also important to consider *multi-directionality.* Multi-directionality refers to the feature of updating behaviors when the behaviors it depends on are updated, but also updating the dependencies when the dependent *behavior* is updated. The interested reader will find a more detailed discussion of these issues in [1].

B. REACTIVE PROGRAMMING À LA ReactiveX

Several of the mainstream reactive tools used by practitioners today adopt similar concepts and abstractions as those found in ReactiveX, namely an Application Programming Interface (API) for asynchronous programming with observable streams of events. In this section, the semantics of ReactiveX is studied, as an alternative to FRP.

Reactive programming as proposed by ReactiveX, provides constructs that operate on discrete values that are emitted over time. The main abstractions in this kind of languages are the *Observables*. An *observable* emits a stream of events (or data) over time, not necessarily at pace. As *behaviors* in FRP, *observables* are composable and new *observables* can be made from simpler ones. Additionally, ReactiveX proposes several operations to act on *observables* allowing them to be filtered, merged, and transformed into new *observables*. As an example, consider the following code:

```
Observable<Integer> varTimeMilliseconds =
        Observable.interval(1,
            MILLISECONDS);
Observable<Integer> varTimeMillisecondsx10
    =
        varTimeMilliseconds.map( ms –> ms
            * 10);
```

The first line declares *observable varTimeMilliseconds* using the *interval* function. Such function creates an *observable* that emits integers starting in *1* and increasing by one unit each millisecond. Then, the second line declares an observable *varTimeMillisecondsx10* using the *map* function to map an anonymous function to multiply by 10 each value emitted by *varTimeMilliseconds*.

In order to define reactions to specific data or event patterns, the ReactiveX languages provide *observers* and an API with the *subscribe* method. The *subscribe* method accepts three functions as parameters. Those functions are bound to three specific stages in the life cycle of observables: *onNext*, *onError*, and *onCompleted*. The function bound to the *onNext* stage is invoked each time that a new value is emitted. The function bound to the *onCompleted* stage is invoked after *onNext* is invoked for the last time, as long as an error is not encountered, which would invoke the function *onError*. More than one *observable* may be attached to a specific stage. As an example, consider the following code:

```
varTimeMillisecondsx10.subscribe(
    s –> print(s + ‘‘has been emitted.’’),
    e –> print(‘‘error occurred: ’’ + e),
    () –> print(‘‘Emission completed.’’));
```

The *subscribe* function receives three lambda functions, and they are bounded respectively to the *onNext*, *onError*, and *onCompleted* stages of the *observable* life cycle. The functions bound to these events are called *observers*.

Regarding the implementation, languages à la ReactiveX suffer from some of the same problems found in FRP. Specifically, *observable* behavior may depend on ‘‘when’’ data is emitted. A ‘‘Hot" *observable* starts emitting data as soon as it is created and, instead of, a ‘‘cold" *observable* starts emitting data as soon as an *observer* is subscribed. ReactiveX and similar languages may also suffer from *glitches*. Naive implementations may produce scenarios where different *observers* receive different streams of data from the same *observable* [22].

III. ReactiveXD: DISTRIBUTED REAL-TIME REACTIVE PROGRAMMING

This section introduces ReactiveXD as a new reactive real-time distributed language. This language is used as an example to discuss implications and implementation constraints.

As will be shown in the state of the art section (Section VI), very few approaches have distribution and time management as first-class citizen concepts. Thus, even though reactive approaches propose a reactive style for concurrency, observables, events, reactions, and behaviors, distribution and time are still treated in a traditional way. Distribution is then controlled using local imperative primitives, and time is only considered implicitly, e.g., as a determinant of the order of events in a computation. These are substantial restrictions in the current context, where users and programmers are thinking of new ways to use and benefit from the ubiquity of the internet, mobile infrastructure, and real-time interaction. Hence,

exploring new ways to incorporate explicit distribution and time management in programming languages seems imperative. Furthermore, several of the abstractions and implementation difficulties presented in the previous sections have more complex and exciting counterparts when considering reactive programming primitives with explicit support for distribution and time manipulation.

To investigate reactive programming with explicit support for distribution and time management, we propose ReactiveXD. ReactiveXD is an object-oriented reactive programming language with explicit support for distribution and explicit management of time. A reactive language with these features would facilitate the design, implementation, and evolution of modern massively distributed and dynamic applications. In this section a minimal set of ReactiveXD features are proposed, while semantics, implementation problems, and design implication are further explored in the rest of the paper.

ReactiveXD has three main concepts: Atomic Distributed Events (ADE), localization and time. ADEs are events occurring at specific nodes in a distributed application. In the paper at hand, an event is atomic to imply that it is the minimal possible event, thus complex events and event patterns are composed of atomic events. Consider, for example, a traditional programming language, there, one may consider method calls and memory access as atomic events. However, a execution method is not considered an atomic event because it may contain several method calls and memory access events. Similarly, a pattern of atomic events, such as a sequence of five events, is not considered an atomic event. These atomic events are the fundamental detectable unit, meaning that ReactiveXD runtime is aware of them and can detect them. But ReactiveXD does not restrict atomic events to local context, the runtime is aware of the distributed topology of the application. Note that several nodes may compose the distributed application, and each node represents an individual computer or a virtual machine inside a physical computer. An event is atomic and distributed to denote that it happens atomically and that remote nodes may detect them, as will be reviewed later in Section IV. In the case of ReactiveXD, the call to a specific method is the only **Atomic Distributed Event**. The information of a distributed event, occurring in a node, is communicated to the other nodes with an **event message** sent through the network.

The second important concept is localization (i.e., space), an application implemented with ReactiveXD can predicate over the localization of a specific ADE and react accordingly. Localization may be absolute (e.g., using a fixed naming scheme like ipv4 addresses) or relative (e.g., using a dynamic naming scheme representing groups of hosts).

The third concept is time, a logical clock (vector clocks) on each node tracks time. Thus all clock readings are local to the node, and no global time is needed. Time in ReactiveXD is always relative. Thus the information of an ADE may arrive at different moments on different nodes. However, using logical clocks, the programmer may predicate over partial orders of events, e.g., she may predicate over a causal relation between two events. As will be seen later, causal relations are only one of the several ways that programmers may address time explicitly in ReactiveXD. For example, they may also use linear temporal logic to address more sophisticated time based predicates.

Let us now present some examples to clarify these main concepts. Consider the following code showing a distributed *observable*:

```
Observable<Event> kp = new
    Observable (call(* UIController.
        keyPressed(*)) && !localhost);
```

The previous code defines a variable *kp* of type *Observable* of events of generic type *Event*. The *Observable* constructor has as a parameter an event predicate expression. The event predicate expression uses a syntax similar to that of aspect-oriented languages. Thus, the constructor will create an observable that emits events matching the expression. Note that this piece of code has distributed semantics already. The expression will match all the method calls to method *keyPressed* on objects of type *UIController* on any host. Additionally, the localization of events is restricted, and the observable *kp* will not emit events happening in the host where the *observable* is deployed. Thus, the code provides enough flexibility to predicate over atomic events, e.g., method calls and event localization.

Let us now consider a more complex piece of code involving explicit time manipulation:

```
Observable<Event> cmkp = new
    Observable (call(* UIController.
        mouseButtonPressed(*))
    && !localhost && !causal);
```

The previous piece of code declares an *observable cmkp* emitting mouse click events happening on remote hosts and that are not causally related. Thus, the events are not only remote, but the expression considers a temporal relation, in this case causality. The causality relation is defined as proposed by Mattern in [17] where two events,*A* and *B*, are causally related if *A* happens before *B* in the same host, if *B* happens after a message notifying event *A* has arrived, or if *B* is causally related to an event *C* that is causally related to*A* (transitivity).

Events that are not causally related are called concurrent, so the predicate `!causal` is `true` whenever the event being evaluated is concurrent with the previous evaluated event (i.e., concurrent means no causal relationships among them). Note that this code already considers time in its semantics.

It is now worth introducing a more complex time related predicate using temporal logic:

```
Observable<Event> cpxObs = new
    Observable (always(kp next(cmpk)));
```

In the previous code, the *observable* will emit an event each time the formula is violated. The formula states that it should be always `true` that: immediately after the observable `kp`

emits an event, the observable `cmkp` will emit an event. This semantics is quite complex but is encoded easily in a predicate using linear temporal logic. ReactiveXD also supports operators such as *eventually*, *weak until* and *strong until*.

ReactiveXD minimal language is big enough to explore the semantics of distributed reactive languages. It already provides advanced features and presents implementation and semantic challenges. For example, ReactiveXD may be affected by distributed *glitch* problems. Ideally, every change in a reactive environment is instantly propagated; nevertheless, due to physical constraints, this behavior hardly ever happens. Reactive non-distributed environments have reached efficient *glitch* management through dependency graphs. However, in a distributed environment a *glitch* is not only affected by naive implementations of updates in the dependency graph, but also by message ordering problems during the propagation of event information in the distributed application. *Glitch* control in the entire system is more complex due to network faults and global clock delays. *Glitches* also cause inefficiencies in applications because a glitch implies a recomputation. Similarly, the evaluation model (push vs. pull) and the multi-directionality have more complicated semantics when considering distributed reactive programming.

Thus, the current set of features already provides a reactive framework capable of detecting complex event behaviors in real-time and reacting accordingly. It is powerful enough to predicate over distributed events and complex relations between them. Explicit time management does not only imply real-time support, but it also implies explicit manipulation of advanced time-based predicates, including detection of causal relations, concurrent events, and detection of relations determined by LTL formulas. Implementing a compiler for such a language is not a trivial task. The semantics must be carefully defined and fine-tuned. The next section introduces a semantic and formal verification tool to facilitate the study and design of reactive real-time language's semantics.

IV. DRRF: A DISTRIBUTED REACTIVE REWRITING FRAMEWORK

This section introduces Distributed Reactive Rewriting Framework (DRRF) as a reactive event-based semantic framework with explicit support for distribution, concurrency and time management. The proposed framework can be used for the specification of executable semantics of reactive distributed languages, and as a formal verification system of programs written using reactive distributed languages.

A. DRRF OVERVIEW

Before presenting the details of DRRF, the general usage and purpose of executable semantics frameworks and formal verification systems will be discussed.

Imagine a programmer willing to create a reactive programming language to address real-time functionality in a distributed application. The programmer may select a set of features to include in the language and proceed to implement a compiler. This is a common methodology to design programming languages, where the semantics and the functionality are directly tested and re-engineered during the compiler's implementation. This methodology uses the original programming language (the one used to implement the compiler) as the primary design artifact. The methodology is also perfectly valid, in the sense that the programming languages, and the abstractions they provide, are powerful thinking and design tools. However, if the programmer detects a problem in the original selection of features or in the semantics enforced by the compiler design, changing such decisions may be too costly and time-consuming. Even worst, if the compiler is already being used for production, the programmer may prefer to keep these wrong decisions to grant backward compatibility with those programs already using the language (this is a common situation in mainstream languages).

Another alternative that the programmer has is to undertake a preliminary design phase using a tool to help her simulate the language's semantics. She may also use the tool to verify properties that she wants to enforce on the programs developed with the programming language. DRRF is one of those tools, providing an environment for formal executable, semantic specification and formal verification of properties. Using DRRF provides at least two advantages, design decisions may be altered and tried with less effort and cost, and of course, the designer will have a formal model of the intended semantics.

Suppose, for example, that the programmer decides to have the following desired features for the programming language: i) observables and observers, ii) explicit distribution, and ii) explicit time constrains detection and manipulation. These previous features are fulfilled by ReactiveXD. She may specify the formal semantics using rewriting logic. She then may use the tool to simulate the programming language (executable semantics), create programs using the language, and verify those programs. She may do all this from scratch or start from a framework such as DRRF that already provides several of the building blocks needed to design distributed reactive real-time programming languages.

1) ABSTRACTIONS PROVIDED BY DRRF

DRRF design metaphor assumes that there is a base distributed application that is monitored by an observing framework and that the observing framework may influence or alter the base application's behavior. Thus, the first thing that must be modeled by DRRF is the base application. A set of processes emitting messages models the base application. Those processes are assumed to be running in independent nodes (computers). The action of emitting a message is considered an event, and these events are tagged with readings of logical clocks (i.e., there is no global clock in the distributed application). Therefore, the framework provides mechanisms to model the processes, the messages, logical time, and the communication mechanism (e.g., unicast vs. multicast).

The second abstraction is the observing framework. In this context, DRRF provides mechanisms to predicate over events occurring in the distributed base application, and mechanisms to specify reactions to those events. DRRF may detect simple atomic events, such as message sending events, or more complex events such as a sequence of events, or a pattern of events related by a time-dependent formula (e.g., an LTL formula). Once an atomic event or a complex event occurs, the observing framework may react and alter the base application's behavior through new emitted events, e.g., calling a method of the application API. DRRF provides abstractions to monitor distributed atomic and complex events and to evaluate complex time-dependent formulas. Note that DRRF does not assume what kind of abstractions the programmer will implement, instead it provides basic building blocks to model reactivity, complex patterns of events, and complex time predicates.

To create such a framework, DRRF was developed in Maude [6], a declarative and reflexive language and system that is based on rewriting logic [18]. In Maude, each computation corresponds to an efficient deduction by rewriting rules [6], [8]. Maude has two kinds of modules: functional modules and system modules. Functional modules correspond to equational theories defining **data types** and **operations** over those types. On the other hand, system modules specify rewriting theories. These theories define data types, equivalence classes, operators, equations and **rewriting rules**.[3] The following sections detail the implementation of the framework in Maude.

B. DESIGN CONSIDERATIONS

The distributed interaction of processes (i.e., not the local sequential semantics of the processes running on each node of the distributed application) is one of the main concerns in the design of DRRF. For example, the interaction through reactive abstractions like observables and observers is one of the aspects to tackle with DRRF. Thus, instead of modeling local sequential semantics, for example, using a shared memory model and a model for sequential processes, we model a working distributed application that may be observed by the reactive abstractions proposed by a given reactive language. Such a language, e.g., ReactiveXD, will monitor distribution, concurrency, and time management explicitly, then, it will create complex and interesting interactions with the base application. Modeling those interactions is the primary purpose of DRRF.

To model the semantics of reactive distributed languages, DRRF includes the following building blocks: base application, communication model, and reactive framework. The base application models the distributed application that will be observed. The distributed base application is modeled as a set of several nodes processing information and communicating with each other via messages. Distribution on the base application may be implemented with an alternative paradigm, i.e., not a reactive language, e.g., it can be implemented through an imperative distributed language. Next, the communication model defines how messages are distributed (e.g., broadcast, multicast, unicast); again, this distribution paradigm is a property of the base application and not of the reactive framework. Finally, the reactive framework provides the core elements and abstractions for observers, reactions, concurrency, distribution, and time management. DRRF provides the elements described above as building blocks, so the language designer may combine and alter them as desired. Similarly, the software engineer may combine and bend these core elements to model and verify properties on distributed applications developed using reactive languages.

The implementation of each building block of the semantic framework is described next.

C. THE BASE APPLICATION

The base application is modeled as a set of concurrent distributed processes. These processes may be of two different types: consumers and producers. Producers emit messages and consumers consume those messages, the emission and consumption of messages are the core events of the distributed application. These events, inside consumers and producers, may be observed by instances of reactive abstractions (e.g., observers).

1) THE SEMANTICS OF CONSUMERS

Each consumer instance is identified with a natural number. Internally, all the consumer instances have the following attributes :

- current-msg: It indicates the message that is being consumed.
- freq-C: It indicates the frequency at which the consumer reads a message. This attribute is invariant throughout the execution.
- actual-size: It represents the number of data blocks that have been consumed from the current message. If the consumer is not consuming any message, the value is 0.
- logs: It saves information about messages that have been consumed.
- clock-c: It stores the value of the vector clock associated with the consumer. A vector clock is modeled with a vector data structure where the individual clock, corresponding to each consumer or producer, is modeled with an integer at a fixed position in the vector. For example, the clock of Consumer 2, is in the second position of the vector. Thus, each time that a consumer receives a message, it compares the vector clock attached to the message with its vector clock and updates the registers accordingly. For a detailed discussion of vector clocks and causality, see [17]).
- limit: It is a numerical value indicating the maximum time for which the consumer is available to consume.

Thus, consumers are modeled as entities consuming messages at a specific pace (frequency) and with a maximum

[3]More details about Made may be found at [6]

capacity of message processing (limit). The consumer clock is modeled using a vector clock, which may allow causality detection. Vector clocks are chosen instead of simple logical clocks to allow a broader range of possibilities for language designers. Moreover, each consumer records information about the messages it has consumed. That information may be used later to predicate over the order of message consumption.or more sophisticated time predicates.

2) THE SEMANTICS OF PRODUCERS

Each producer instance also is identified with a natural number, and the producer instances have the following attributes:

- cnt-prod: The number of messages issued by the producer.
- frequency: This attribute is invariant throughout the execution and indicates how often the producer sends a message.
- sizes: This attribute is invariant throughout the execution and notifies the number of data blocks associated with the messages issued by the producer.
- Notice-logs: Log of messages produced and consumed.
- clock-p : It stores the value of the vector clock associated with the consumer. A vector clock is modeled with a vector data structure where the individual clock corresponding to each producer or consumer is modeled with an integer at a fixed position in the vector. For example, the clock of **producer** 1, is in the first position of the vector. Thus, each time that a producer emits a message, it updates its vector clock and attaches the vector clock reading to the message. For a detailed discussion of vector clocks and causality, see [17]).
- end: The maximum number of messages that can be issued by the producer.

D. TIME MANAGEMENT

Vector clocks [17] are used as the main abstraction for time management (see [28] for a detailed discussion of time management on distributed applications). Hence, every message issued is labeled with a vector clock reading. That reading identifies the moment where the message was issued. Each node in the distributed system has a vector clock.

One of the most relevant feature to use from vector clocks is the detection of the causal relation. This feature allows programmers to take into account relationships of order among distributed messages. Figure 1 shows a possible behavior of a distributed system with three processes, $\{P_1, P_2, P_3\}$. The vector clock of the process *P3* when the event $e_{3,3}$ occurs (notice that the event $e_{i,j}$ refers to the event number j of the process i) allows to infer that the process *P3* is causally related to processes *P2* and *P3*, however, in the event $e_{3,2}$ the same inference may not be done.

Now, using MAUDE specification, the Module **VECTOR** was defined to implements vector clocks. The clocks are lists of natural numbers, where the length of the list represents the number of processes present in the distributed application.

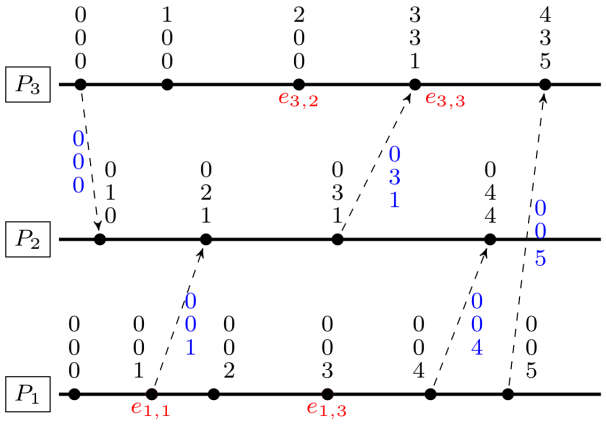

FIGURE 1. Time model of a distributed system composed by 3 processes.

A piece of the module which defines vector clocks is shown below.

```
1  mod VECTOR is
2    pr NAT . pr LIST{Nat} .
3    sort Vector .
4    subsorts Nat < List{Nat} < Vector .
5    op tick : Vector Nat -> Vector .
6    op _<v_ : Vector Vector -> Bool .
7    op maxv : Vector Vector -> Vector .
8    op ic : Vector Nat -> Nat .
9     ...
10 endm
```

The first line of **VECTOR** initializes the module. The second line indicates the use of the module of natural numbers (**NAT**) and a list of natural numbers . Line 4 specifies the relation between natural numbers, the natural numbers list and the vector. The following lines (5 - 8) define the operations for managing vector clocks. The operation _ < **v** _ defines the order of the vectors, the operation **maxv(V,V')** compares two vectors and the operation **ic(V,N)** returns the state of the specific vector clock V at the position N.

E. COMMUNICATION MODELS

The semantics of communication models is now defined following ideas presented in [24]. The base application may implement one of these four models: broadcast, multicast, unicast, unordered unicast. This section describes the implementation of unordered unicast and broadcast semantics.

1) UNORDERED UNICAST MESSAGES

With unordered unicast semantics, the producers emit messages without specifying the emitter or the recipient. Thus, any producer may emit a message, and any consumer may consume any message. The base application is modeled by producers producing massages and consumers consuming those messages. Each message is produced once and consumed only once. Below, the formal message definition in

```
mod BROADCAST is
  op msg_from_to_ : Tuple Nat Nat -> Msg [ctor] .
  op multicast_from_to_ : Tuple Nat Set{Nat} -> Configuration .
  op broadcast_from_ : Tuple Nat -> Msg [ctor] .
  ...
endm
```

FIGURE 2. Definition of the Broadcast module.

Maude for the communication model is presented. In the code, the operation **Unordered-msg** defines a constructor for messages of this type. The constructor receives as parameters three natural parameters and a vector. It then creates a unique object of type **Msg**. The natural numbers in the data structure represent the id of the message; the size of the message; the frequency of the producer that emitted the message; and the vector clock reading of the producer when the message was sent.

```
op Unordered–msg : Nat Nat Nat Vector
    -> Msg.
```

2) BROADCAST MESSAGE

The Broadcast communication model is implemented in three stages as follows. First, the producer creates a message of type**broadcast_ from _**, which contains a tuple with the identifier of the producer, the size of the message, and the vector clock representing the vector clock of the producer when the message was emitted (a piece of code of the Broadcast module is shown in figure 2). Then, once the message is emitted, the dynamic rewriting rules will create a second message of type **multicast_from_to**, which is similar to the original one but has an additional field with the set of identifiers of all consumers in the system. Finally, utilizing the rewriting rules, the system creates a unicast message for each consumer in the application, specifying the origin and corresponding destination, using the type message **msg _ from _ to _**.

The following code shows a rewriting rule transforming messages of type **broadcast_from_** into messages of type **multicast_from_to**:

```
rl [broadcasting]:
{ (broadcast ((N,T,TS)) from PR)
  (Names| NA |) Con }
=>
{ (multicast ((N,T,TS)) from PR to NA )
(Names| NA |) Con }.
```

The rule specifies that if the configuration contains: a message of type broadcast _from_, a registry with the identifiers of all consumers (Names NA) and other objects (Con), it will create a new configuration deleting the broadcast_from_ message, adding a new message of type multicast _from_to, and leaving the registry and the other objects unaltered.

F. THE DYNAMIC BEHAVIOR OF THE BASE APPLICATION

As the static elements of the model (Producers, Consumers, Messages) were previously described, this section presents the dynamic model. In Maude, the dynamics of the model will be specified using rewriting rules. Those rules are declarative, and they specify how to rewrite the configuration if a specific pattern is present. The model can not predict the order of applicability of the rewriting rules, and instead, when the system is model checked, it will apply the rewriting rules in all possible orders, generating and evaluating several paths.

To model the dynamic behavior of the base application, it is essential to specify how messages are produced and emitted and how clocks are altered. In DRRF, there is not a centralized global clock. Instead, each consumer and producer has a clock. Producers emit messages in one tick of the clock, and consumers may use several ticks to consume a message, being the number of required ticks dependent on the size of the message that is being consumed.

Each vector clock advances in an independent way simulating a decentralized model. In the decentralized model, the vector clock of each node advances while the rest of the clocks in the system remain fixed. This behavior is modeled using the rewriting rules **[update-time-Producer]** and **[update-time-Consumer]** for Producers and Consumers, respectively.

One Producer can send a message if it is in the correct frequency, and it does not exceed the maximum messages allowed. In this way, the rule **[send-msg]** may change the data flow adding a new message to the configuration. This message becomes available for any Consumer. A Consumer can consume messages at the specified pace (frequency) without exceeding the maximum indicated limit. The rule **[Consumer-msg]** reads a message and stores it in the **current-msg** attribute. The rule **[Consuming-msg]** models the consumption of information of a message that is inside a Consumer. The message will be consumed during a fixed number of ticks, taking into account the internal clock of the consumer. When a message is completely consumed, the rule **[end-msg]** is responsible for releasing the **current-msg** attribute so that the consumer continues reading available messages.

Figure 3 shows the code for the initial state of a reactive distributed application modeled in Maude with 2 Producers and 3 Consumers. The Producers have different production frequencies, and their message limits are the same. In this way, $Producer_1$ will produce more messages than $Producer_0$,

```
1 {< 0 : Producer | cnt-prod: 0,frequency: 1,clock-p: (0 0 0 0 0),sizes: 5,end: 10,finish: false,Notice-
      logs: nil >
2 < 1 : Producer | cnt-prod: 0,frequency: 3,clock-p: (0 0 0 0 0),sizes: 1,end: 10,finish: false,Notice-logs
      : nil >
3 < 2 : Consumer | current-msg: none,freq-C: 2,actual-size: 0,logs: nil,limit: 34, clock-c:(0 0 0 0 0) >
4 < 3 : Consumer | current-msg: none,freq-C: 1,actual-size: 0,logs: nil,limit: 30, clock-c:(0 0 0 0 0) >
5 < 4 : Consumer | current-msg: none,freq-C: 5,actual-size: 0,logs: nil,limit: 55, clock-c:(0 0 0 0 0) >}
```

FIGURE 3. Initial state of a reactive distributed application in Maude.

and at the end of the execution, $Consumer_3$ will be the one that has consumed the most messages. On the other hand, each consumer has different time limits.

The most relevant rewriting rules for an unordered unicast model are described next.

1) Rule **[update-time-producer]**

```
1 crl [update-time-Producer] :
2 { < PR : Producer | ATS, clock-p:
      CL, end: END, cnt-prod: CT' >
      Con }
3 =>
4 { < PR : Producer | ATS, clock-p:
      tick(CL,PR), end: END, cnt-
      prod: CT' > Con }
5 if CT' < END.
```

the rule states that each producer will increase its clock independently by one. In this case the details of increasing the vector clock by one are hidden by the operation **tick**. Message emission does not depend on the tick of the clock, instead it depends on the clock value and the specified frequency.

The code display the implementation of the rule.The keyword **crl** indicates the initiation of a conditional rule with the condition shown in line 5. This rule is only applied if the condition is true. In this case, only if the number of emitted messages (**CT′**) is below the specified maximum (**END**). The lines between 4 and 5 define the rule, specifying the configuration before and after applying the rule.

2) Rule **[send - msg]**

```
1 crl [send-msg]:
2     { < PR : Producer | ATS, cnt-
        prod: N, weights: T,
        frequency: FR, clock-p:
        TS, end: END > Con }
3     =>
4     { < PR : Producer | ATS, cnt-
        prod: s(N), weights: T,
        frequency: FR, clock-p:
        tick(TS,PR), end: END >
        Unordered-msg(PR,N,T,TS)
        Con }
5 if FR divides ic(TS,PR) /\
6     (END > N) /\
7     (ic(TS,PR) =/= 0).
```

The rule models the emission of a message. The implementation uses a conditional rule stating that the rule will only be applied if the value of the individual vector clock corresponds to an integer (greater than 0) multiple of the frequency, the maximum number of messages has not been emitted yet and the producer clock is already ticking (i.e., has started its computations). Once the conditional is validated, the *Producer* sends a message and increments its internal clock by one (one tick). Just after the rule **[send-msg]** is applied, an **Unordered-msg(PR,N,T,TS)** is added to the configuration where **PR** represents the identifier of the Producer, **N** designates the number of messages, **T** represents the number of ticks required to get the message consumed, and **TS** is the value of the vector clock when the message was emitted. Note that the clock value attached to the message is the one before the tick.

Lines 2 and 7 specify that the rule **[send-msg]** will be applied if a Producer satisfies the proposed restrictions. Line 6 – 7 indicates the condition to be evaluated. The Producer must have a permitted frequency to emit **FR divides ic(TS,PR)** and must not have exceeded the message limit **END > N** initializing the vector clock **ic(TS,PR) =/= 0**. Finally, line 4 specifies the configuration after applying the rule where the argument **cnt-prod** and the producer vector clock are increased by one unit, and the message is finally attached.

3) Rule **[consumer-msg]**

```
1 crl [Consumer-msg]:
2     {Unordered-msg(PR,N,T,TS)
3     < CS : Consumer | ATS,current
        -msg:none,freq-C: FR,
        clock-c: CL,limit: LT >
        Con }
4     =>
5     {< CS : Consumer | ATS,
        current-msg:Unordered-
        msg(PR,N,T,TS), freq-C: FR,
        limit: LT
6     clock-c: tick(maxv(CL,TS),CS)
        > Notice-msg(CS,PR,tick
        (maxv(CL,TS),CS)) Con }
7 if (FR divides ic(CL,CS))
8     /\
9     ic(CL,CS) <= LT.
```

A consumer may consume a message when the value of the individual vector clock corresponds to an integer that is a multiple of the frequency, and additionally the maximum number of consumed messages have not been reached yet. Additionally, when a consumer reads a message, a notification is issued to the DRRF aiming to inform the producer about the consumption of the event, and then the consumer vector clock is updated.
The rule **[Consumer-msg]** implementation, shown in the code above, specifies a message reception event. Lines 1 to 3 define the name of the rule and the triggering event. There, the triggering event is the message defined in the previous rule (**[send-msg]**). The message **Unordered-msg(PR,N,T,TS)** is taken from the configuration with **PR** representing the identifier of the Producer, **N** designating the number of messages, **T** representing the number of ticks required to get the message consumed, and **TS** containing the value of the vector clock on the producer when the message was emitted. Lines 5 to 7 contain the actions after the rule is applied. At line 5 the attribute **current-msg** gets the message and at the same line the operation **maxv** updates the consumer vector clock. After the application of the rule, the vector clock is consistently updated. The conditions at lines $9-11$ are similar to the conditions presented before for the rule **[send-msg]**, nevertheless consumers are particularly limited in terms of consumption by its vector clock.

G. REACTIVE FRAMEWORK

Section II introduced ReactiveXD, a reactive programming language with explicit support for distribution and time management. The language provides observables, atomic distributed events, localization, and time-aware predicates. With such building blocks explained above, we can now specify the executable semantics of the language. To introduce the semantics, a series of experiments addressing several abstractions and mechanisms found in ReactiveXD will be described.

1) OBSERVABLES AS LTL FORMULAE

Observables may be defined using linear-time temporal logic (LTL) predicates, which generate events each time the LTL formula is violated. In the DRRF implementation, the module **LTL** defines the temporal logic operators as Maude operations. With the aim of giving the truth value for an LTL formula in a particular configuration, the operation **verification** is defined as having the following attributes: an LTL formula and a specific configuration where that formula needs to be evaluated. The operation **verification** is defined as follows:

```
1 op verification: Ltl Configuration ->
      Ltl.
```

The operation **verification** recursively defines the semantics of temporal logic using the LTL semantics, the author in [10] provides detailed discussion of LTL semantics. An example of LTL semantics is the formula $[]\phi$ ([] means "Always"), which is **false** if there is a moment during the computational trace where ϕ is **false**, otherwise, it is **true**. In contrast, the formula $E\phi$ (E means eventually in the future) is **true** when at any moment ϕ is **true**, and it is **false** if during the entire execution ϕ is never **true**. These two LTL formulas are modeled in Maude within the operation **verification**, replacing ϕ by an LTL predicate, as shown next:

```
1 eq verification([](LT),Con) =
2      if (verification(LT,Con) == false)
3       then false
4       else [](LT) fi .
5 eq verification(E(LT),Con) =
6      if verification((LT),Con)
7       then true
8       else E(LT) fi .
```

2) OBSERVERS

Observers (also called checkers) are defined by the module **SYSTEM-CHECK** and use a LTL formula to monitor specific behaviors. The identifier of the observable determines the association of an observer with an observable (Consumer or Producer). A checker and the observable it is associated with must have the same identifier (remember that in Maude, the identity of and object depends on its type and its identifier.) When the LTL formula that the checker verifies is not fulfilled, a reaction can be generated, if such a reaction has been defined. Figure 4 shows an example of the association between an Observer ($checker_0$) and an Observable ($Producer_0$).

3) USING THE verification OPERATION

```
1 crl [checker-Predicate] :
2 { < RC : checker | Formula: TL, finish:
      false, count: N, end-c: END > Con }
3 =>
4 { < RC : checker | Formula:
      verification(TL,Con), finish:
      false, count: s(N), end-c: END > Con }
5 if N <= END.
```

The code presented above illustrates how the dynamic behavior of the operation **verification** is defined. The behavior of observers is modeled through the rewriting rule **[checker-Predicate]** allowing to verify the behavior of the associated observer. In DRRF, this rule is modeled as shown in line 4 with the operation **verification** that has as a parameter an LTL formula. Additionally, the maximum number of steps that are allowed in the computational trace is defined by the attribute **end-c**, as this is a required limit to have an accurate evaluation of **Always** and **Eventually** LTL predicates.

4) DEFINITION OF PREDICATES TO GUARD BEHAVIOR

The module **PREDICATES** defines all the predicates required to specify behaviors. The code shown below defines the basic operation **|-**, defined within the module.

```
1 {< 0 : Producer | cnt-prod: 0,frequency: 1, clock-p: (0 0 0), sizes: 7, end: 8, finish: false, Notice-logs:
      nil >
2 < 0 : checker | count: 0, end-c: 1000, Formula:true, finish: false > }
```

FIGURE 4. Association between a Producer and a Checker.

```
1 op _|-_ : Sys Ltl -> Ltl .
2 op _Causality_ : Nat Nat -> Pred .
3 ---Causaly detection over
      Consumers
4 ceq{< RC:Consumer|ATS,clock-c: VC~>
      Con}|-(RC Causality SE)=true if ic(
      VC,SE) > 0 .
5 eq {Con}|-(RC Causality SE)=false[
      owise].
```

Such operation is used to define when a predicate is **false** or **true**. At line 4 **ceq** marks the start of a conditional equation claiming that the clock of the individual **VC** (Vector Clock) at position **SE** must be greater than 0. Line 5 specifies that if the previously defined condition is not fulfilled, the value of the predicate will be **false**. To illustrate, the operation can be used to define the predicate **Causality** that checks if there is a causal relationship between consumers **SE** and **RC**.

Now the formal model for ReactiveXD is complete. It has been modeled in DRRF using observables (modeled in the **CONSUMER** and **PRODUCER** modules), and observers that have been modeled in the **SYSTEM-CHECK** module. Additionally, the dynamic behavior of the reactive application, as well as the data flow represented by the communication models (Broadcast or Unordered Unicast), have been specified by the module **RULES**. Note that the specification even allows glitch detection using predicates over causal relations.

V. EVALUATION

DRRF provides a configurable formal framework capable of specifying different executable semantics of reactive distributed languages and implementing verification tools for the specified languages. DRRF is also capable of analyzing the dynamic behavior of distributed reactive applications and particularly the verification of specific execution properties. One advantage of analyzing distributed reactive systems using a formalism-based framework, such as DRRF, is the possibility of simulating the system execution before implementing it. The simulation of the execution ensures the verification of properties mitigating unexpected behaviors that would come up during a real execution.

The following three scenarios show how DRRF may be used when designing reactive languages and distributed applications. The first scenario presents the specification of an executable semantics for ReactiveXD configured with causality detection (vector clocks), and temporal logic predicates. This scenario shows how LTL predicates may be used to detect complex patterns in a simple distributed application using broadcast communication as the primary mechanism for message distribution. The second scenario uses the same specification of ReactiveXD to show how to predicate over and verify causality properties in an application where the message order gets disrupted. The last scenario models how ReactiveXD may be used to detect multi-step security attacks in an IoT distributed system with multiple nodes.

A. SIMULATING A ReactiveXD APPLICATION WITH A BROADCAST COMMUNICATION MODEL

The first scenario, shown in figure 5, simulates a ReactiveXD application with a broadcast communication model. The scenario presents a producer and three consumers. The producer broadcasts a message to each consumer. Messages are emitted in a specific order (first the black message and then the white message), and they may be received in a different order by each consumer. An observer with an LTL formula is observing a stream of events from the producer. Note that the stream of events may include message sending events and message reception events, but they may also involve other events, e.g., memory updates. Consumers will acknowledge via a message when they have received a message from the producer.

To specify such scenario in DRRF, it must be first configured with suitable components. In this case, DRRF has been configured with the semantics of ReactiveXD, namely vector clocks, casual order, temporal logic predicates, distributed semantics and a broadcast communication model for the base application (i.e., the application that may be observed). After the static configuration with the initial conditions was defined, the scenario to simulate and verify properties specified in the LTL formula was executed.

Figure 6 presents an example of an initial state for a ReactiveXD application with a broadcast communication model. This application is composed of four observables (one Producer and three Consumers) and one observer. The latter is subscribed to one of the Producers ($Producer_0$). Note that $Observer_0$ contains the formula **E(Order 0)** in line 8, and **Order 0** is a predicate defined in the **PREDICATES** module those values **true** where the $Observer_0$ finds at least two messages that have been consumed in the wrong order, and **false** otherwise. Moreover, the $Observer_0$ will monitor if there is an error in the order of data consumption at the Observable ($Producer_0$).

Then, the model is executed and properties violations are checked. When the command **SEARCH** is executed, more than 650 states were found, where the data generated by $Observable_0$ was being consumed in an incorrect order. For example, one error is presented in figure 7. Note that the

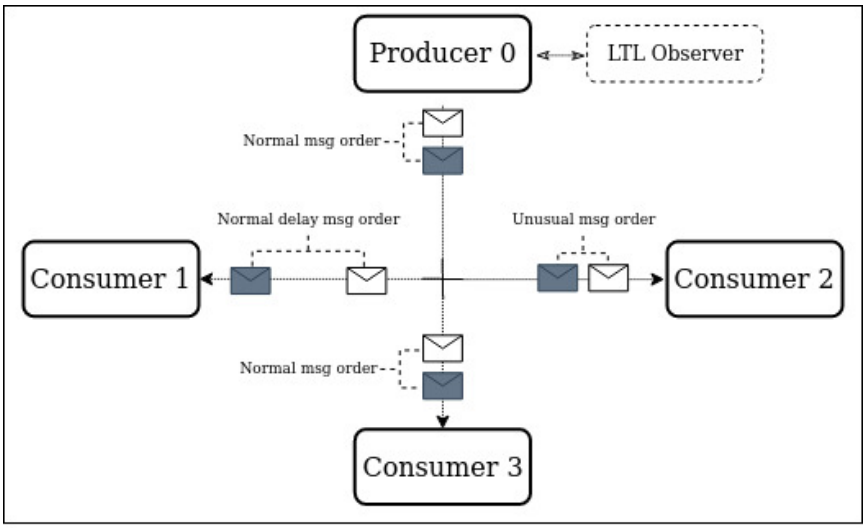

FIGURE 5. Representation of a ReactiveXD application with broadcast communication model.

```
1  search in SYSTEM-CHECK :
2  ---Definition of the Initial State in lines 3-10
3  {
4  < 0 : Producer | cnt-prod: 0, frequency: 3,clock-p: (0 0 0 0), weights: 1, end: 6, finish: false, Notice-
       logs: nil >
5  < 1 : Consumer | current-mesg: none, freq-C: 2, actual-time: 0, logs: nil, limit: 50, clock-c:(0 0 0 0 ) >
6  < 2 : Consumer | current-mesg: none, freq-C: 1, actual-time: 0, logs: nil, limit: 50, clock-c:(0 0 0 0 ) >
7  < 3 : Consumer | current-mesg: none, freq-C: 1, actual-time: 0, logs: nil, limit: 50, clock-c:(0 0 0 0 ) >
8  < 0 : Observer | Formula: (E (Order 0)) , finish: false, count: 0 , end-c: 50 >
9    Names| (1,2,3) |
10 }
11 =>*
12 {Con < 0 : checker | ATS , Formula: true >}
13 .
```

FIGURE 6. The initial state for a ReactiveXD application with a broadcast communication model.

attribute Notice-logs of $Observable_0$ in line 5 contains interesting information about the reception of messages by all the Consumers. Particularly, Notice-logs shows the evolution of the system:

- Message 0 was received by Consumer 1, in the time 5 of the Producer clock and in the time 3 of the Consumer clock.(|0; 5; 1; 3|)
- Message 0 was received by Consumer 2, in the time 7 of the Producer clock and in the time 4 of the Consumer clock.(|0; 7; 2; 4|)
- Message 1 was received by Consumer 2, in the time 8 of the Producer clock and in the time 1 of the Consumer clock.(|1; 8; 2; 1|)

From these logs, one error in the order of consumption of messages by $Consumer_2$ was detected, for instance, $message_1$ was read by $Consumer_2$ in time 1 (on the consumer's internal clock) while $message_0$, generated before than $message_1$, was consumed much later, 4 ticks later. This case reveals an issue in the communication (delay, interruption, re-transmission, etc.) that caused $message_1$ to be consumed before $message_0$. Line 7 in figure 7 shows this error, there attribute current-mesg indicates that the current message is number 0, while the attribute logs indicated that the previous message was message number 1.

B. SIMULATING A ReactiveXD APPLICATION WITH AN UNICAST COMMUNICATION MODEL

Reactive languages can be used to find states that break FIFO (first-in, first-out) consistency, as researchers in [15] have shown it. For example, imagine a distributed application with one component sending messages and other components receiving messages, as shown in figure 8, imagine then that two observers are deployed in different nodes to check that messages arrive in the order they were emitted. Now assume the following scenario, one observer detects errors in the order of message reception, but the other one detects the message order correctly. Thus one of them has seen the incorrect sequence (a FIFO consistency violation). ReactiveXD can be

```
1  Solution 565 (state 3145851)
2  states: 3145852 rewrites: 118006946 in 1862000ms cpu (1866417ms real) (63376 rewrites/second)
3  Con —>
4  Names| 1, 2, 3 |
5  < 0 : Producer | cnt—prod: 2,frequency: 3,clock—p: (9 3 4 0),weights: 1,end: 6,finish: false,Notice—logs
       : (| 0 ; 5 ; 1 ; 3 | | 0 ; 7 ; 2 ; 4 | | 1 ; 8 ; 2 ; 1 |) >
6  < 1 :Consumer | current—mesg: msg(0, 0, 1, 3 0 0 0),freq—C: 2,actual—time: 0,logs: nil,limit: 50,clock—c:
       (3 3 0 0) >
7  < 2 : Consumer | current—mesg: msg(0, 0, 1, 3 0 0 0),freq—C: 1,actual—time: 0,logs: "0_1_1",limit: 50,
       clock—c: (6 3 4 0) >
8  < 3 : Consumer | current—mesg: none,freq—C: 1,actual—time: 0,logs: nil,limit: 50,clock—c: (0 0 0 0) >
9  msg (0,1,3 0 0 0) from 0 to 3)
10 msg (1,1,6 3 0 0) from 0 to 1
11 msg (1,1,6 3 0 0) from 0 to 3
12 ATS —> finish: false,end—c: 50,count: 1
```

FIGURE 7. Error found for the ReactiveXD application with the broadcast communication mode.

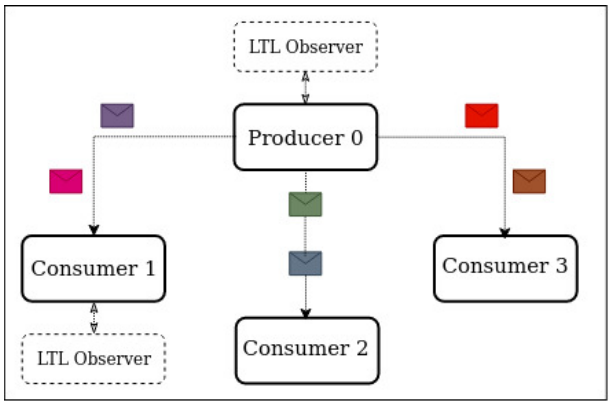

FIGURE 8. Representation of a ReactiveXD application with a unicast communication model.

used to implement observers for such applications and detect the FIFO consistency violation. Next, DRRF will be used to model this full scenario.

Figure 9 shows the specification of the initial state and operational rules for a ReactiveXD application, with a unicast communication model, implementing the scenario described above. In order to implement this model, engineers just need to use the model for unicast communication, and this changes how communication is done. Nonetheless, the semantics and syntax of the base application and the Observables are still the same. This can be seen by comparing figures 6 and 9 where the communication model is significantly different, but the base application does not require any change. The specification in DRRF shows $Producer_0$ sending individual messages to each *Consumer*, and observers looking for a consistency violation. The predicates Order 0 of 1 and Order 1 of 0, in lines 7 and 8, supports the FIFO consistency validation. Predicate Order 0 of 1 enables $Observer_0$ to verify if $Observable_0$ is consuming messages in a different order from the order of emission at $Observable_1$. Similarly, predicate Order 1 of 0 enables $Observer_1$ to verify if $Observable_1$ is consuming messages in the same order as when they were emitted by $Observable_0$.

Once the specification is ready, it is executed in DRRF. The Maude SEARCH command simulates all possible computation states looking for reachable states where $Observer_0$ detects an error in the consumption of messages coming from $Observable_1$ (i.e. predicate E(Order 0 of 1) evaluates true) but $Observable_1$ detects a proper consumption. After the execution, a FIFO consistency violation was found to be breached in several states, having one of them shown in figure 10. Line 5 contains the attribute Notice-logs of $Observable_0$, which has useful information about the reception of messages by all the Consumers. In this case Notice-logs states the following:

- Message 0 was received by Consumer 1, in the time 4 of the Producer clock and in the time 5 of the Consumer clock.(|1; 3; 1; 1|)
- Message 1 was received by Consumer 1, in the time 3 of the Producer clock and in the time 1 of the Consumer clock.(|0; 4; 1; 5|)

The information contained in Notice-logs allows DRRF to conclude that the predicate Order 0 of 1 is fulfilled and an error in the order of consumption of messages by $observable_1$ has taken place. On the other hand, the predicate Order 1 of 0 in line 6 has not been fulfilled, indicating that the order of message consumption of $Observable_1$ from $Observable_0$ was correct. So, the order in which $Observable_1$ consumed the messages from $Observable_0$ was correct but $Observable_0$ detected the contrary. Therefore, in this case, the FIFO consistency condition was breached.

As shown in examples presented in sections V-A and V-B, DRRF can be used to verify the behavior of ReactiveXD applications, to simulate communication models, and to simulate violation of properties in the application. Note that not only consistency violations are being detected through the formal specification, but actual constructs of ReactiveXD are being modeled to make predicates over time constraints. Even, new predicates may be defined to grant

```
search in SYSTEM-CHECK :
{
< 0 : Producer | cnt-prod: 0,frequency: 1,clock-p: (0 0 0 0 0),weights: 2,end: 7,finish: false,Notice-
    logs: nil >
< 1 : Consumer | current-mesg: none,freq-C: 2,actual-time: 0,logs: nil,limit: 20, clock-c:(0 0 0 0 0 ) >
< 2 : Consumer | current-mesg: none,freq-C: 1,actual-time: 0,logs: nil,limit: 20, clock-c:(0 0 0 0 0 ) >
< 3 : Consumer | current-mesg: none,freq-C: 5,actual-time: 0,logs: nil,limit: 20, clock-c:(0 0 0 0 0 ) >
< 0 : checker | Formula: (E (Order 0 of 1)) , finish: false , count: 0 , end-c: 50 >
< 1 : checker | Formula: (E (Order 1 of 0)) , finish: false , count: 0 , end-c: 50 >
}
=>*
{Con < 0 : checker | ATS , Formula: true >
 < 1 : checker | ATS , Formula: (E Order 1 of 0) >
```

FIGURE 9. The initial state for a ReactiveXD application with a unicast communication model.

that a distributed reactive application makes adequate and efficient prevention of glitches.

C. SIMULATING AN IoT ECOSYSTEM THROUGH ReactiveXD

IoT ecosystems generally have different components on the user side, e.g., IoT devices, IoT sentinels, among others. These systems also have server-side components, generally deployed on the cloud, e.g., IoT Hub, analytic servers, among others. All these components interact and exchange information among them. Thus, an IoT ecosystem can be considered a distributed reactive system with a heterogeneous set of components, where each component performs specific activities that are essential to creating a complex IoT ecosystem. As IoT is gaining more presence in different and new contexts, it has become mandatory to design ways to protect all the data that is managed by the different components of an IoT ecosystem. Therefore, the proposal presented in the paper at hand can be applied in defense of the IoT ecosystems helping to:

- Detect and prevent attacks over existing IoT services deployed in production environments, revealing situations or traces that would not be considered by conventional attack detection methods.
- Improve the design of a forthcoming IoT service through the detection of failures in the data consumption by IoT components.

To model an IoT ecosystem in ReactiveXD, it is necessary to develop a set of modules that represent the dynamic behavior of the ecosystem itself (the base application). Additionally, in this scenario, DRRF is used to simulate how ReactiveXD may be used to detect multi-step attacks in the IoT ecosystem. A multi-step attack follows a predefined pattern based on different consecutive phases, for example:

1) An IoT device accesses an Android market.
2) A malicious IoT mobile application is downloaded to the IoT device.
3) The malicious IoT mobile application drops a payload over the IoT device.
4) The payload executes a vulnerability scanning of the entire IoT infrastructure to detect vulnerable servers that are accessible only by authorized devices.
5) The payload performs a command injection over one of the vulnerable IoT servers.

To model an IoT multi-step attack in ReactiveXD, the module Multi-step-attack [4] was implemented specifying the dynamic behavior of an IoT ecosystem. It should be clarified that in this specification, the Consumers and Producers, i.e., the base application, are components of the IoT ecosystem. An IoT device, such as an intelligent light bulb receiving control messages from an IoT server, is modeled in ReactiveXD as a Consumer. In the same way, an IoT device that sends messages toward an IoT server, such as a temperature sensor reporting values, is modeled as a Producer. At last, an IoT device that sends and receives messages can be modeled as Consumers and Producers with the same identifier. Using ReactiveXD, a programmer specifies a security attack pattern using LTL formulas or causal predicates. Observables deployed in IoT ecosystem verify these formulas. An illustration of this scenario is shown in Figure 11.

This scenario was then simulated on DRRF. Consider an IoT ecosystem composed of five Observables and one Observer. This scenario has an initial state defined as shown in figure 12. In order to model the multi-step attack, new objects are included in this configuration. The *Server* objects at lines 8-9 model an intrusion detection system and a host-based intrusion detection system server found in the IoT provider's cloud. Likewise, *Marker* objects represent the market place where the IoT devices access to download the application (for a more detailed information of multi-step security attacks see in [4]). Note, how simple is to define the new semantics and elements for a particular scenario on top of the base modules of DRRF. In particular, the semantics of the attack was defined in the module MULTI-STEP-ATACK, which uses the base application and observer modules.

The tree of the formula presented in the *Observer* at line 11 is shown in Figure 13. This formula detects a

[4]https://masanar.github.io/DRRF/index.html

```
Solution 2 (state 2454597)
states: 2454598 rewrites: 92187651 in 830948ms cpu (830950ms real) (110942 rewrites/second)
Con
—>
< 0 : Producer | cnt—prod: 2,frequency: 1,clock—p: (5 5 0 0 0),weights: 2,end: 7,finish: false,Notice—
      logs: (| 1 ; 3 ; 1 ; 1 | | 0 ; 4 ; 1 ; 5 |) >
< 1 : Consumer | current—mesg: Unordered—mesg(0, 0, 1, 5 0 0 0 0),freq—C: 2,actual—time: 0,logs: | 0 ; 0 ;
      2|,limit: 20,clock—c: (2 5 0 0 0) >
< 2 : Consumer | current—mesg: none,freq—C: 1,actual—time: 0,logs: nil,limit: 20,clock—c: (0 0 0 0 0) >
< 3 : Consumer | current—mesg: none,freq—C: 5,actual—time: 0,logs: nil,limit: 20,clock—c: (0 0 0 0 0) >
ATS —> finish: false,end—c: 50,count: 1
```

FIGURE 10. Error identified for a ReactiveXD application with a unicast communication model.

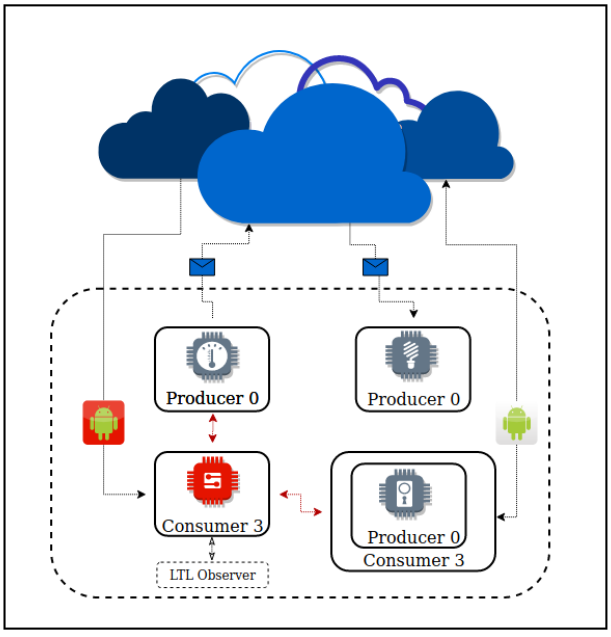

FIGURE 11. Representation of an IoT ecosystem.

multi-step attack. For this reason, the predicates defined within the MULTI-STEP-ATACK module, have the following semantics:

- Acces-market-D(0): Whenever the $Observable_0$ access to the market, this predicate evaluates True.
- App-download-D(0): Is True if the $Observable_0$ download an app from the market, notice that in a multi-step attack, this happens after access to the market.
- Payload-D(1): Whenever the $Observable_1$ gets a payload due to an app download, the predicate is True.
- Scanning-D(1): Is True whenever the $Observable_1$ is already infected with a payload, and this starts scanning the IoT ecosystem.

In this sense, the LTL formula specifies the following behavior: always that the $Observable_0$ access to the market, and eventually the $Observable_0$ download an app, the $Observable_1$ gets a payload then eventually this payload start scanning the IoT ecosystem. In other words, this formula is True whenever an attack is taking place. At that moment, the $Oberver_0$ may react by sending a shutdown message to the IoT device. Note that the specification already has distributed semantics and predicates over distributed messages. Starting from the previous initial configuration, DRRF can check different executions for the IoT system until it detects traces that evidence that a multi-step attack may happen. An application designer may then act accordingly and correct the system to protect it from multi-step attacks.

VI. STATE OF THE ART

This section summarizes related work from two different perspectives, namely programming languages and formal modeling. First, functional reactive languages, reactive languages à la ReactiveX, and distributed reactive programming are introduced, highlighting its differences with ReactiveXD. Next, formal frameworks that have been used to investigate the semantics of reactive distributed programming languages are presented, comparing those approaches with DRRF.

A. FUNCTIONAL REACTIVE LANGUAGES

Functional Reactive Programming (FRP) emerged as a framework to program 3D computer graphics, providing abstractions to model entities with continuous time-changing attributes. The ability to model the dynamic behavior of distributed global systems with heterogeneous architectures promoted the research of FRP tools in other domains, and several implementations were proposed. Fran [9] (Functional Reactive Animation), the first functional reactive approach, provides an extensive collection of elements to model interactive multimedia objects using behaviors and events. Behaviors are time-varying and reactive values, while events model concrete conditions happening at discrete times. Fran Provides rich semantics based on Haskell, allowing the composition of complex behaviors from primitive ones. However, to the best of our knowledge, no LTL libraries or causality libraries have been incorporated into Fran. Similarly, in Frappé [7], authors implement FRP in Java, integrating the behaviors/event model with Java Beans' event/property

```
{
< 0 : Producer | cnt-prod: 0,frequency: 1,clock-p: (0 0 0 0 0 0),weights: 2,end: 8,finish: false,Notice-
    logs: nil >
< 1 : Consumer | current-mesg: none,freq-C: 1,actual-time: 0,logs: nil,limit: 70,clock-c: (0 0 0 0 0 0) >
< 2 : Producer | cnt-prod: 0,frequency: 2,clock-p: (0 0 0 0 0 0),weights: 3,end: 5,finish: false,Notice-
    logs: nil >
< 3 : Consumer | current-mesg: none,freq-C: 1,actual-time: 0,logs: nil,limit: 20,clock-c: (0 0 0 0 0 0) >
< 4 : Consumer | current-mesg: none,freq-C: 1,actual-time: 0,logs: nil,limit: 10,clock-c: (0 0 0 0 0 0) >
< 5 : Producer | cnt-prod: 0,frequency: 1,clock-p: (0 0 0 0 0 0),weights: 1,end: 7,finish: false,Notice-
    logs: nil >
< 5051 : Market | Requets: nil >
< 8030 : Server | Devices: 1,HIPS: false,NIPS: false >
< 8050 : Server | Devices: 1,HIPS: false,NIPS: false >
< 0 : Observer | count: 0, end-c: 1000, Formula: ( [](Acces-market-D(0))->( (E App-download-D(0)) -> ( (
    E Payload-D(1)) -> ( (E Scanning-D(1)) ) ) ) ) , finish: false >
}
```

FIGURE 12. The initial state for an IoT ecosystem modelled with ReactiveXD.

model, making any java component available to the FRP framework. However, Frappé has several constraints, e.g., the events (represented with Java Beans) are synchronously processed by a single-thread, generating dependency on a single process. Moreover, Frappé lacks functionality to model sophisticated time predicates such as LTL predicates or causal relations. Flapjax [20] provides behaviors (continuous time-varying abstractions) and event-driven reactive evaluation. It was written as a Javascript library and can be used on top of any javascript framework. As the previous approaches, its main features are related to event reactivity and not to the explicit and complex manipulation of time predicates.

Functional reactive programming has also been used in specialized domains. For example, Procera [30] provides a declarative language to express network policies at a high-level reactive. It was designed to respond to the dynamic changes happening in a traditional network. Procera supports events at the level of switches, access permissions, bandwidth, and resource consumption during network traffic. Another example of the applicability of FRP is Frob [25] (Functional Robotics), this Haskell-based domain-specific language for robot control provides abstractions such as behaviors and reactivity components. Even though these implementations exploit functional reactive programming, they do not provide complex time predicates or explicit distribution.

The languages and frameworks discussed above provide abstractions similar to the original Fran's behaviors and implement reactivity with events. However, their applicability is limited to scenarios with simple time requirements, for example, reacting in real-time to discrete events. In contrast, ReactiveXD includes mechanisms to model complex time predicates, supporting, for example, Linear Temporal Logic (LTL) predicates, and causal predicates, involving ordered patterns of several discrete events. Thus, traditional functional reactive approaches react to atomic events or simple sequences of events, such as `Event-1` after `Event-1`, but reactions to complex time relations such as causality, or real parallelism, or temporal logic predicates, is outside its repertoire. Furthermore, ReactiveXD is aware of the distributed environment and may predicate over event localization. Current functional reactive approaches react to local events (events in its own machine) but ignore events occurring in other nodes. When programmers want to react to remote events, they must implement manually the functionality to make such remote events available as local events.

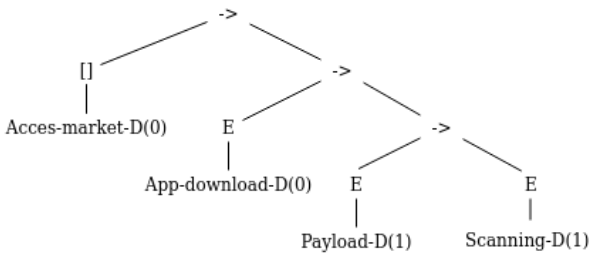

FIGURE 13. LTL aimig to detect a multi-step attack.

B. LANGUAGES AND FRAMEWORKS Á LA ReactiveX

Reactive Extensions (ReactiveX) was born around 2010, designed by the computer scientist Erik Meijer at Microsoft.[5] It proposes a model that consists of Observables that emit data, a set of operators to modify the data, and Observers that consume or "watch" the data. ReactiveX and similar approaches [21], [27] propose then a flexible framework extending the observer pattern. The extensibility comes from the composition operators allowing compositions and explicit manipulation of the streams of events emitted by observables. However flexible, the operator framework provides only limited capabilities regarding event compositions and time manipulation. Consider for example the operators `And` and `join`. The `And` operator receives two or more observables and creates a new one that emits events each time that the original observables provide an event, thus creating an opportunity to react to a specific relation among the individual events.

[5]http://reactivex.io/

This mechanism is flexible, but quite limited and primitive, complex relations must be coded explicitly in an imperative way in the source code. Instead, using ReactiveXD, the programmer may create explicit declarative predicates to express intended relation among events. Similarly, the `Join` operator provides means to combine events from observables whenever an event from one observable is emitted during a time window of an event emitted by other observable. This construct allows programmers to react to time constrains, however, these mechanisms are not as expressive as causal predicates or LTL predicates.

The ideas presented in this paper regarding ReactiveXD may be ported to other languages. The semantics are not attached or derived from a specific programming language. Any programming language supporting object orientation may be extended with the ideas presented here. However, the implementation of a concrete compiler and a runtime is still a subject of future work (some preliminary work may be found in [4]). The algorithms written in ReactiveXD have a more declarative form and may be used for reactive programming on heterogeneous architectures, as the IoT examples presented here. It may also be be used, for example, for the instrumentation of big-data middleware.

C. DISTRIBUTED REACTIVE PROGRAMMING

Applying reactive abstractions to distribute programs is called Distributed Reactive Programming. As in the sequential case, there is no consensus on the semantics of a distributed reactive framework. For the present discussion we will classify the approaches in two groups: weak distribution support and strong distribution support. We will first present some of their characteristics, and then we will compare them with ReactiveXD.

Approaches with weak distribution support provide a reactive framework where distribution is treated as a second-class citizen. Thus, the approaches may react to events generated in remote environments, but the framework itself does not provide the means to make those events available to the reactive components. Instead, some external component provides distribution support, i.e., distribution is handled with imperative primitives. Most of the reactive languages presented above are of this type. Most of them may react to distributed events, but those events are generated with non-reactive constructs (see for example [7], [21], [27]).

On the other hand, approaches with strong distribution support do consider distribution as a first-class citizen in the language. For example, in [22], authors propose to maintain a decentralized dependency graph. In this case, the main concern, and the main difficulty, is maintaining the information of a distributed dependency graph consistently. It is particularly difficult to cope with the dynamic behavior of the system, handling for example new nodes in the topology.

ReactiveXD presents strong distribution support. However, we do not compute the dependency graph at runtime, but instead, we compute what atomic events are made available to interested observers at compile time. Note that we make available only events of interest to the observers, thus, achieving efficient distribution of remote events. This feature makes the approach more scalable and fault-tolerant without sacrificing reactivity or efficiency.

Furthermore, none of the approaches discussed above provides the means to model complex events or time-dependent predicates as proposed in our reactive distributed language. However, language designers of those reactive frameworks may use DRRF to model their frameworks including reactions to complex event patterns augmented with time-dependent predicates.

D. SEMANTIC FRAMEWORKS FOR REACTIVE LANGUAGES

Rewriting logic [19], [26] has been proposed as a logic able to represent concurrent systems, distributed systems, and programming languages. It has been used to model executable semantics for concurrent programming languages and derive their corresponding formal analysis tools. It has also been used to specify concurrency models and distributed algorithms. All these features make rewriting logic a suitable formalism to investigate the semantics of reactive languages, including functional reactive languages and languages like ReactiveXD. Furthermore, as modeling concurrency and distribution explicitly in a language was one of the concerns of the paper at hand, it was clear that a formalism was required to support such characteristics naturally. Such flexibility has been widely studied, see for example [24], where authors show how rewriting logic can be used to model distributed dynamic systems (systems which architecture may change during time) and reason about the concurrent changes that occur in the system.

Other authors have addressed the study of semantics for reactive distributed frameworks explicitly. For example, using the DREAM middleware, researchers explore in [16] different semantics for distributed reactive systems qualitatively. Furthermore, in [15] the authors present a study of the change propagation cost on DREAM's API and its consistency implementation on the Java language. However, to the best of our knowledge, DRRF is the first configurable and executable semantic framework able to model several of the main aspects of reactive languages, including the functional reactive variants and à la ReactiveX variants. DRRF provides a richer set of features, including support for LTL predicates, explicit distribution, and explicit causal predicates.

Other formalisms have been investigated as adequate means to model concurrent real-time systems [3], [5], [11], [14], including models for distributed and concurrent computations. Those models have been accompanied by verification tools that simulate and verify the behavior of the specified model. In [3], the authors use algebraic processes to define the semantics of an aspect-oriented language with explicit distribution [2]. Nevertheless, this approach generates a big gap with the implementation and does not provide mechanisms to model causality or LTL predicates. Similarly, Tabareau proposes a semantic framework for distributed aspects using join calculus [29]. This proposal serves as a specification

framework to test the properties of the woven application; however, to the best of our knowledge, there is no verification tool available. Future work may investigate the suitability of those approaches to investigate the semantics of ReactiveXD, including the explicit manipulation of time and distribution awareness.

VII. CONCLUSION

This paper explores the semantics of distributed reactive real-time languages proposing DRRF, a semantic framework for the design of distributed reactive real-time languages and applications. The paper studies first the state of the art of reactive languages identifying two trends: functional reactive languages and reactive languages à la ReactiveX. Functional reactive languages can manipulate objects with continuous time-changing attributes (e.g., 3D animation objects). Reactive language à la ReactiveX extends the observer pattern with composable reactions, addressing the real-time evaluation of streams of discrete events (languages à la ReactiveX). The latter (languages à la ReactiveX) are currently mainstream tools used widely for the implementation of web front ends and microservices. From this study, it is concluded that several features of reactive programming are worth studying to address standard requirements presented in the current global dynamic and heterogeneous computer systems. In particular, it is argued that for the manipulation of real-time constraints, reactive tools must have the flexibility of languages like ReactiveX and powerful time-related abstractions as those presented in functional reactive tools. Thus, the paper at hand proposes ReactiveXD, an event-based language with explicit support for distribution, concurrency, and time management. This language provides a syntax similar to that of ReactiveX but is augmented with time-aware predicates using Linear Temporal Logic and causal abstractions. The paper also proposes DRRF, a framework based on rewriting logic and implemented on top of Maude, that serves as a formal specification framework for the semantics of distributed time-aware reactive languages and also serves as a verification tool for applications implemented on those languages. Finally, an evaluation of the applicability of DRRF is performed by implementing several scenarios, including a scenario for the detection of a multi-step attack on an IoT ecosystem.

This work shows the flexibility of DRRF, rewriting logic, and Maude to specify executable semantics of distributed frameworks and programming languages. It also shows how easily a language designer may propose sophisticated semantics and test them before creating a full implementation of the compiler. ReactiveXD was proposed and enriched with a robust set of features to manipulate localization and time explicitly. Such features are not trivial and were tried and tested on an IoT scenario to demonstrate its usefulness.

VIII. FUTURE WORK

As future work, we propose to improve the specification so that its execution can be more efficient, avoiding the explosion of states. We plan to research on the detection of new types of attacks in IoT networks using as a base the implementation of the IoT scenario developed in this paper. Particularly, we plan to investigate the way to integrate ReactiveXD into security implementations for IoT that: i) are currently focused on the application of IoT sentinels to protect local scenarios, and could be improved with the support for distribution management for the detection of traffic anomalies or security incidents that exist in distributed and spread scenarios, and ii) are focused on the protection of security events generated in IoT scenarios using a blockchain-based architecture, and could be improved with the support of predicates for time management for the detection of Advanced Persistent Threats (APT).

REFERENCES

[1] E. Bainomugisha, A. L. Carreton, T. V. Cutsem, S. Mostinckx, and W. D. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 52:1–52:34, Aug. 2013, doi: 10.1145/2501654.2501666.

[2] L. D. Benavides Navarro, R. Douence, and M. Südholt, "Debugging and testing middleware with aspect-based control-flow and causal patterns," in *Proc. 9th Int. Middleware Conf.*, Leuven, Belgium: Springer-Verlag, Dec. 2008.

[3] L. D. Benavides Navarro, R. Douence, A. Núñez, and M. Südholt, "LTS-based semantics and property analysis of distributed aspects and invasive patterns," in *Proc. Workshop Aspects, Dependencies Interact.*, vol. 517, K. U. Leuven, Ed. Belgium, Jul. 2008, pp. 36–45. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00469648

[4] L. D. B. Navarro, C. Pimienta, M. Sanabria, D. Díaz, W. Garzón, W. Melo, and H. Arboleda, "REAL-T: Time modularization in reactive distributed applications," in *Advances in Computing*, J. E. Serrano and J. C. Martínez-Santos, Eds. Cham, Switzerland: Springer, 2018, pp. 113–127.

[5] G. Bhat, R. Cleaveland, and G. Lüttgen, "A practical approach to implementing real-time semantics," *Ann. Softw. Eng.*, vol. 7, no. 1, pp. 127–155, Oct. 1999.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude—A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. New York, NY, USA: Springer-Verlag, 2007.

[7] A. Courtney, "Frappé: Functional reactive programming in Java," in *Proc. Int. Symp. Practical Aspects Declarative Lang.* Cham, Switzerland: Springer, 2001, pp. 29–44.

[8] F. Durán, C. Rocha, and J. M. Álvarez, "Towards a maude formal environment," in *Formal Modeling: Actors, Open Systems, Biological Systems*. Cham, Switzerland: Springer, 2011, pp. 329–351.

[9] C. Elliott and P. Hudak, "Functional reactive animation," in *Proc. 2nd ACM SIGPLAN Int. Conf. Funct. Program. (ICFP)*, 1997, pp. 263–273. [Online]. Available: http://conal.net/papers/icfp97/

[10] M. Fisher, *An Introduction to Practical Formal Methods Using Temporal Logic*. Hoboken, NJ, USA: Wiley, 2011.

[11] P. Fontana and R. Cleaveland, "A menagerie of timed automata," *ACM Comput. Surv.*, vol. 46, no. 3, pp. 40:1–40:56, Jan. 2014, doi: 10.1145/2518102.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming Language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.

[13] P. Haller and H. Miller, "Ray: Integrating RX and ASYNC for direct-style reactive streams," in *Proc. Workshop Reactivity, Events Modularity*, 2013, pp. 1–7.

[14] J. Magee, *Concurrency: State Models & Java Programs*. Hoboken, NJ, USA: Wiley, 2006.

[15] A. Margara and G. Salvaneschi, "On the semantics of distributed reactive programming: The cost of consistency," *IEEE Trans. Softw. Eng.*, vol. 44, no. 7, pp. 689–711, Jul. 2018, doi: 10.1109/tse.2018.2833109.

[16] A. Margara and G. Salvaneschi, "We have a DREAM: Distributed reactive programming with consistency guarantees," in *Proc. 8th ACM Int. Conf. Distrib. Event-Based Syst. (DEBS)*, 2014, pp. 142–153, doi: 10.1145/2611286.2611290.

[17] F. Mattern, "Virtual time and global states of distributed systems," *Parallel Distrib. Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.
[18] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 73–155, Apr. 1992.
[19] J. Meseguer, "Twenty years of rewriting logic," *J. Logic Algebraic Program.*, vol. 81, nos. 7–8, pp. 721–781, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1567832612000707
[20] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: A programming language for ajax applications," *ACM SIGPLAN Notices*, vol. 44, no. 10, pp. 1–20, 2009.
[21] A. Mosteo, "RxAda: An Ada implementation of the ReactiveX API," in *Reliable Software Technologies—Ada-Europe*, J. Blieberger and M. Bader, Eds. Cham, Switzerland: Springer, 2017, pp. 153–166.
[22] F. Myter, C. Scholliers, and W. De Meuter, "Distributed reactive programming for reactive distributed systems," *Art, Sci., Eng. Program.*, vol. 3, 2019. [Online]. Available: https://programming-journal.org/2019/3/5/
[23] M. Nischt, H. Prendinger, E. André, and M. Ishizuka, "MPML3D: a reactive framework for the multimodal presentation markup language," in *Proc. Int. Workshop Intell. Virtual Agents*. Cham, Switzerland: Springer, 2006, pp. 218–229.
[24] P. C. Ölveczky, "Modeling Distributed Systems in Rewriting Logic," in *Designing Reliable Distributed Systems*. London, U.K.: Springer, 2017, doi: 10.1007/978-1-4471-6687-0.
[25] J. Peterson, P. Hudak, and C. Elliott, "Lambda in motion: Controlling robots with haskell," in *Proc. Int. Symp. Practical Aspects Declarative Lang.* Cham, Switzerland: Springer, 1999, pp. 91–105.
[26] G. Roşu, "From rewriting logic, to programming language semantics, to program verification," in *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer* (Lecture Notes in Computer Science), vol. 9200. Cham, Switzerland: Springer, 2015, pp. 598–616.
[27] G. Salvaneschi, A. Margara, and G. Tamburrelli, "Reactive programming: A walkthrough," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 953–954.
[28] M. Sanabria, W. G. Alfonso, and L. D. B. Navarro, "Towards real-time semantics for a distributed event-based mop language," in *New Trends in Model and Data Engineering*, E. H. Abdelwahed, L. Bellatreche, D. Benslimane, M. Golfarelli, S. Jean, D. Mery, K. Nakamatsu, and C. Ordonez, Eds. Cham, Switzerland: Springer, 2018, pp. 231–243.
[29] N. Tabareau, "A theory of distributed aspects," in *Proc. 8th Int. Conf. Aspect-Oriented Softw. Develop. (AOSD)*, New York, NY, USA, 2010, pp. 133–144, doi: 10.1145/1739230.1739246.
[30] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control," in *Proc. 1st Workshop Hot Topics Softw. Defined Netw.*, 2012, pp. 43–48.

MATEO SANABRIA-ARDILA received the B.Sc. degree in math from the Escuela Colombiana de Ingeniería Julio Garavito, in 2018. He is currently pursuing the master's degree in computer sciences with the Universidad de Los Andes, Bogotá, Colombia. He is also an Instructor Professor with the Escuela Colombiana de Ingeniería Julio Garavito. His current research interests include formal methods, mathematical logic, cryptography, and computability theory and algorithms.

LUIS DANIEL BENAVIDES NAVARRO (Member, IEEE) received the degrees in software construction and electrical engineering from the Universidad de Los Andes, Colombia, the master's degree in computer science from Vrije Universiteit Brussel, Belgium, and the Ph.D. degree in computer science from the University of Nantes, France. He is currently an Associate Professor with the Escuela Colombiana de Ingeniería Julio Garavito. His research interests include enterprise architecture, software engineering, distributed computing, distributed programming languages, and models for the development of distributed, complex, and concurrent applications.

DANIEL DÍAZ-LÓPEZ received the Ph.D. degree in computer science from the University of Murcia, Spain. He is currently an Assistant Professor with the School of Engineering, Science and Technology, Universidad del Rosario, Colombia. He has experience in management of technological infrastructure for data centers, design of secure communications networks, and implementation of information security management systems. His research interests include techniques for cybersecurity, cyber intelligence, privacy-preserving mechanisms, secure software development lifecycle, ethical hacking, and security for the IoT.

WILMER GARZÓN-ALFONSO received the B.Sc. degree in computer science, in 2006, the B.Sc. degree in mathematics from the Escuela Colombiana de Ingeniería, in 2008, and the Master of Science degree in computer engineering from the University of Puerto Rico. His master's thesis was in bioinformatics. He has over ten years, implementing technological projects focused on automatization of process. He is currently a Part-Time Professor with the Escuela Colombiana de Ingeniería Julio Garavito. His research interests include data analytics, data mining and learning techniques, distributed systems applications, and formal methods.